Assisting Static Compiler Vectorization with a Speculative Dynamic Vectorizer in a HW/SW Co-designed Environment

RAKESH KUMAR†, UPC Barcelona
ALEJANDRO MARTÍNEZ†, Intel Barcelona Research Center, Intel Labs
ANTONIO GONZÁLEZ†, Intel Barcelona Research Center, Intel Labs and UPC Barcelona

Compiler based static vectorization is used widely to extract data level parallelism from computation intensive applications. Static vectorization is very effective in vectorizing traditional array based applications. However, compilers inability to do accurate interprocedural pointer disambiguation and interprocedural array dependence analysis severely limits vectorization opportunities. HW/SW co-designed processors provide an excellent opportunity to optimize the applications at runtime. The availability of dynamic application behavior at runtime helps in capturing vectorization opportunities generally missed by the compilers.

This paper proposes to complement the static vectorization with a speculative dynamic vectorizer in a HW/SW co-design processor. We present a speculative dynamic vectorization algorithm that speculatively reorders ambiguous memory references to uncover vectorization opportunities. The speculative reordering of memory instructions avoids the need of accurate interprocedural pointer disambiguation and interprocedural array dependence analysis. The hardware checks for any memory dependence violation due to speculative vectorization and takes corrective action in case of violation. Our experiments show that the combined (static + dynamic) vectorization approach provides 2x performance benefit compared to the static GCC vectorization alone, for SPECFP2006. Furthermore, speculative dynamic vectorizer is able to vectorize 48% of the loops that ICC failed to vectorize due to conservative dependence analysis in TSVC benchmark suite. Moreover, dynamic vectorization scheme is as effective in vectorization of pointer-based applications as for the array-based ones, whereas compilers lose significant vectorization opportunities in pointer-based applications. Furthermore, we show that speculation is not only a luxury but also a necessity for runtime vectorization.

Categories and Subject Descriptors: **C.1.2 [Computer System Organization]**: Multiprocessor-SIMD; **D.3.4 [Software]**: Processors-Optimization

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Hardware/Software Co-designed Processors, Vectorization, Speculation, Dynamic Optimizations

ACM Reference Format:
Rakesh Kumar, Alejandro Martínez, and Antonio González. 2016. Assisting static compiler vectorization with a speculative dynamic vectorizer in an HW/SW codesigned environment. ACM Trans. Comput. Syst.33, 4, Article 12 (January 2016), 33 pages.

This work is partially supported by the Generalitat de Catalunya under grant 2009SGR-1250, the Spanish Ministry of Education and Science under grant TIN 2010-18368, and Intel Corporation.

This article extends an earlier version, "Speculative Dynamic Vectorization to Assist Static Vectorization in a HW/SW Co-designed Environment" [Kumar et al. 2013] that appeared in the 20th International Conference on High Performance Computing (HiPC 2013).

Author's addresses: R. Kumar (corresponding author) University of Edinburgh, Edinburgh, United Kingdom, email: rkumar2@inf.ed.ac.uk; A. Martínez, ARM Ltd. Cambridge, United Kingdom, email: Alejandro.MartinezVicente@arm.com; A. González, Department of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, Spain, email: antonio@ac.upc.edu

† R. Kumar is now with School of Informatics, University of Edinburgh. A. Martínez is with ARM Ltd. Cambridge and A. González is with Department of Computer Architecture, UPC Barcelona.

1. INTRODUCTION

Single Instruction Multiple Data (SIMD) accelerators form an integral part of modern microprocessors. These can be found in processors from different computing domains like general purpose processors [Intel Software Developer´s Manual; Diefendorff et al. 2000; Lee 1996], Digital Signal Processors [D´Arcy et al. 1999], gaming consoles [Kahle et al. 2005; Sporny et al. 2002] as well as embedded architectures [Baron 2005]. SIMD accelerators are tailored to exploit data level parallelism from modern multimedia, scientific and throughput computing applications. Since these accelerators perform the same operation on multiple pieces of data, they just require duplicated functional units and a very simple control mechanism. Due to this simplicity, SIMD accelerators grow in size with each new generation. For example, Intel´s MMX [Intel Software Developer´s Manual] had a vector length of 64-bits, which was increased to 128-bits in SSE extensions [Intel Software Developer´s Manual]. Intel´s recent SIMD extension AVX [Intel Software Developer´s Manual] and Intel´s Xeon Phi [Intel Xeon Phi Coprocessor] supports 256-bit and 512-bit vector operations respectively.

Code generation for SIMD accelerators has always been challenging. In the early days, programmers used to target these accelerators mainly using in-line assembly or specialized library calls. Then, automatic generation of SIMD instructions (auto-vectorization) was introduced in compilers [Naishlos 2004; Bik et al. 2002], which borrowed their methodology from vector compilers. These compilers target loops for generating code for SIMD accelerators. S. Larsen [Larsen et al. 2000] introduced Superword Level Parallelism (SLP) in which they target basic blocks instead of whole loops for vectorization. These static approaches to vectorization are effective for traditional applications where memory is referenced through explicit array accesses, whereas modern applications make extensive use of pointers. The inability of compilers to do accurate interprocedural pointer disambiguation and interprocedural array dependence analysis limits the vectorization opportunities in the current and future applications [Maleki et al. 2011].

In this paper, we propose to have dynamic vectorization as a complementary optimization to the compiler based static vectorization. It is important to note that we do not propose to eliminate static vectorization altogether because there are several complex and time consuming transformations which are not straightforward to apply at runtime and are too costly like loop distribution, loop interchange, loop peeling, memory layout change, algorithm substitution etc. However, static vectorization alone fails to capture significant vectorization opportunities due to conservative memory disambiguation analysis. To handle these cases we propose to have a speculative dynamic vectorizer which can speculatively reorder ambiguous memory references, without any need for accurate interprocedural pointer disambiguation and interprocedural array dependence analysis, to uncover vectorization opportunities. Moreover, in the absence of loops, the scope of vectorization for static vectorization is a single basic block. We propose to vectorize bigger code regions which include multiple basic blocks and can be created at runtime following the biased direction of branches.

Furthermore, we propose a speculative dynamic vectorization algorithm which can be implemented in the software layer of a HW/SW co-designed processor[1]. The proposed algorithm speculatively reorders and vectorizes memory operations. The speculative reordering of memory instructions avoids the need of accurate

[1] Section 2 provides background about HW/SW Co-designed Processors.

interprocedural pointer disambiguation and interprocedural array dependence analysis. During execution, the hardware checks for any memory dependence violations caused by speculative vectorization. If any violation is detected, the hardware rolls back to a previously saved check-point and executes a non-speculative version of the code. The hardware support required for speculative execution is already provided by co-designed processors like Transmeta Crusoe [Dehnert et al. 2003], BOA [Sathaye et al. 1999] etc. Therefore, no additional hardware support is needed from speculative vectorization point of view. This hardware support is also one of the reasons for choosing HW/SW co-designed processors over dynamic binary optimizers in our proposals.

Moreover, in the absence of static compiler vectorization, our algorithm can work as a standalone vectorizer also. Therefore, the legacy code that was not compiled for any SIMD accelerator can be vectorized using the proposed algorithm. The co-designed nature of the processor makes the vectorization portable. As a result, the algorithm can be modified to transparently target a different SIMD accelerator. It is important to note that the proposed algorithm does not require any compiler or operating system support/modification. The main contributions of this paper can be summarized as:

(1) Proposes a complementary dynamic vectorization technique that does not require interprocedural pointer disambiguation and interprocedural array dependence analysis.
(2) Proposes to increase the vectorization scope utilizing the dynamically discovered control flow: biased branch directions and dynamic loop trip counts.
(3) A runtime speculative vectorization algorithm :
    - that is equally good in vectorizing array and pointer based applications.
    - that is able to vectorize legacy code.
(4) Experimental evaluation of the proposed algorithm and it's comparison with GCC and ICC vectorizers.
(5) A study of different components of dynamic instruction stream to gain more insights about effectiveness of vectorization and limiting factors.
(6) A study to understand the importance of speculation in the runtime vectorization and its robustness.

The rest of the paper is organized as follows: Section 2 provides a background on HW/SW co-designed processors. Section 3 briefly provides the motivation for the work presented in this paper. Section 4 describes the proposed algorithm with an example. Section 5 explains the speculation and recovery mechanism. Evaluation of the algorithm using Test Suite for Vectorizing Compilers (TSVC), SPECFP2006, Physicsbench and UTDSP applications is presented in Section 6. Section 7 presents the related work and Section 8 concludes.

2. BACKGROUND OF HW/SW CO-DESIGNED PROCESSORS

A HW/SW co-designed processors is a hybrid architecture that leverages hardware /software co-design to couple a software layer to the microarchitectural design of a processor. The software layer resides between the hardware and the operating system. This software layer allows host and guest ISAs to be completely different by translating the guest ISA instructions to the host ISA dynamically. We define the host ISA as the ISA that is implemented in the hardware, whereas, guest ISA is the one for which applications are compiled. The basic idea behind these processors is to have a simple host ISA to reduce power consumption and complexity. This kind of

processors [Ebcioğlu et al. 1997; Sathaye et al. 1999; Dehnert et al. 2003] have enticed researchers for more than a decade. Moreover, there is a renewed interest in them in both industry and academia [Intel's HW/SW co-designed processor project; Lupon et al. 2014; Branković et al 2014; Branković et al 2013; Wang et al. 2013; Pavlou et al 2012; Neelakantam et al 2010].

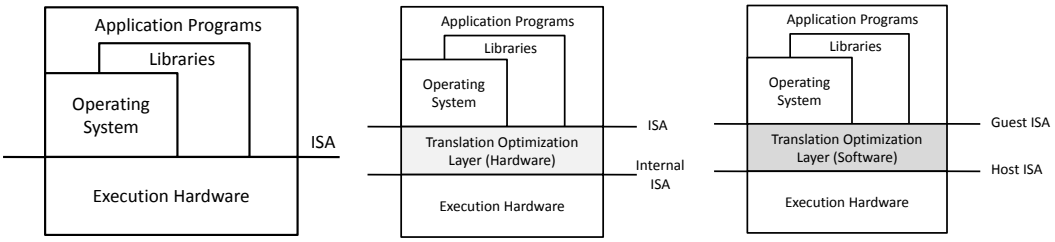
These processors are specifically designed to achieve energy efficiency, design simplicity, and performance improvement. In order to achieve design simplicity, they keep the hardware simple and implement a relatively simple ISA. The simple hardware design also helps in achieving energy efficiency. Transmeta reports significant reduction in power dissipation for their HW/SW co-designed processor Crusoe compared to Intel Pentium III for a software DVD player [Klaiber 2000]. Their data shows that Pentium III heats up to a temperature of 105º C whereas Crusoe´s maximum temperature goes only up to 48º C running the same software DVD player. Furthermore, to achieve the performance goal, HW/SW co-designed processors employ dynamic binary optimizations.

In general, HW/SW co-designed processors implement a proprietary ISA in order to achieve design simplicity and power efficiency. Therefore, they need to apply binary translation to map the guest ISA on to the host ISA. The binary translation can be implemented in either hardware or software. Modern processors implementing CISC ISA, like x86, implement binary translation in hardware [Smith et al. 2005]. The hardware binary translator translates CISC instructions to RISC like instructions dynamically to simplify the execution pipeline implementation. However, the hardware implementation leads to significant hardware complexity and power consumption. HW/SW co-designed processors, on the other hand, implements dynamic binary translation in software which leads to power efficiency.

Figure 1a shows the hardware/software interface in a conventional RISC processor where the software stack directly interacts with the hardware. Conventional CISC processors implement a RISC like ISA in hardware. As shown in Figure 1b, they employ a hardware dynamic binary translator to translate CISC instructions to the internal ISA instructions. The binary translation in HW/SW co-designed processors is performed by a software layer as shows Figure 1c. We call this software layer as Translation Optimization Layer (TOL) in this paper.

Performing the dynamic binary translation/optimization in software layer provides several benefits over the hardware implementation. For example, the software implementation significantly reduces hardware complexity and power consumption. Furthermore, it allows to upgrade a processor in the field by introducing new optimizations in the software layer. On the contrary, if TOL is implemented in hardware, adding new optimizations in the existing processor is not feasible. Additionally, software implementation of TOL significantly reduces hardware validation and verification cost and time.

In HW/SW co-designed processors, TOL resides in a ROM and is the first program to start execution when system boots up. Since TOL acts as an insulation



a) Conventional RISC processor. b) Conventional CISC processor. c) HW/SW co-designed processor.

Figure 1 HW/SW interface in processors.

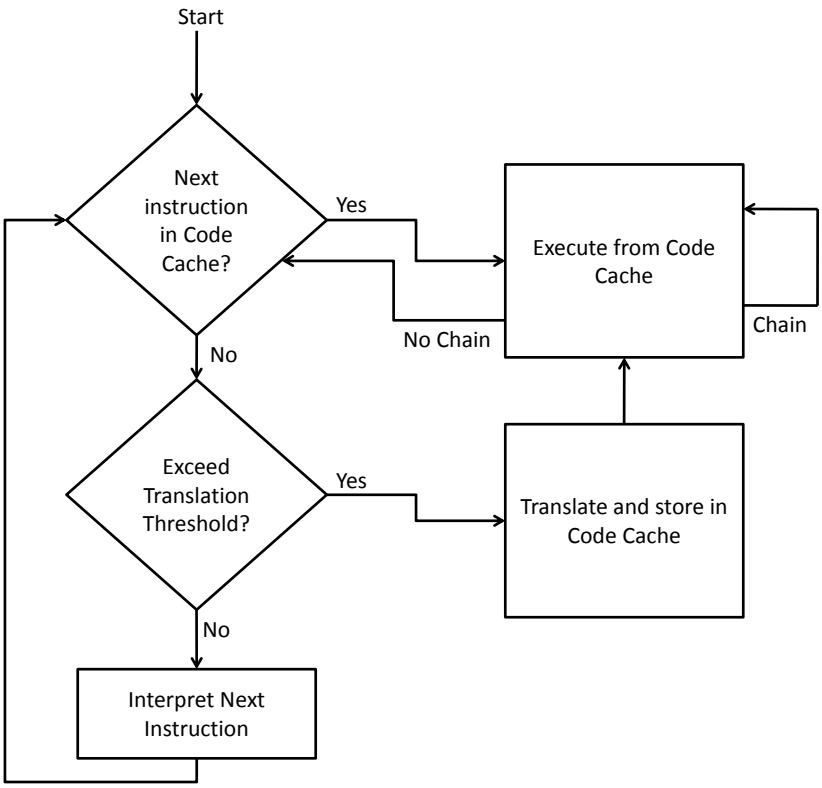


Figure 2 Typical two stage TOL control flow.

layer between the conventional software stack and the hardware, the host ISA can be changed arbitrarily without having to make changes in the conventional software stack. The only modification needed in this case would be to have a new version of TOL that translates guest ISA code to the new hardware. Since the execution of TOL itself requires some processor time, it might affect the overall performance. However, in addition to binary translation, TOL is also responsible for optimizing the translated binary to boost the performance and compensate for its own execution overhead.

2.1 Binary Translation/Optimization

As said before translating guest ISA code to host ISA is the prime responsibility of TOL. The translation is done dynamically and generally, in multiple phases. Usually, in the first phase, an interpreter decodes and executes guest ISA instructions sequentially. In the rest of the phases, the guest code in translated into host ISA code and stored in the code cache, after applying several dynamic optimizations, for faster execution. The number of translation phases and optimizations in each phase are implementation dependent.

Figure 2 shows a typical two stage translation/optimizations flow in a TOL. It starts by interpreting guest ISA instruction stream sequentially. While interpreting, TOL also profiles the guest code to collect information about most frequently executed code and biased branch directions. The execution frequency guides TOL to decide which guest code basic blocks to translate. When a basic block has been executed more than a predetermined number of times, TOL invokes the translator. The translator takes the guest ISA basic blocks as input, translates them to host ISA

code and saves the translated code into the code cache for fast native execution. Instead of translating and optimizing each basic block in isolation, the translator uses biased branch direction information, collected during interpretation, to create bigger optimization regions, called superblocks. A superblock, generally, consists of multiple basic blocks following the biased direction of branches. Therefore, superblocks increase the scope of optimizations to multiple basic blocks and allow more aggressive optimizations. Superblocks have a single entry point that is the first instruction of the first basic block included in the superblock. However, depending on the implementation they might have multiple or a single exit point, making them a single-entry multiple-exit or single-entry single-exit structure.

Initially, the control is transferred back to TOL after executing a superblock from the code cache. Then, TOL searches the next instructions to be executed. If the next instruction is not already translated, it has to be interpreted. However, if it is already translated, TOL patches the last branch of the first superblock (the one that transferred the control back to TOL) to the beginning of the second superblock. This process is called chaining [Dehnert et al. 2003] or linking [Bala et al. 2000]. Chaining enables the control to be transferred directly from one superblock to the other without having to come back to TOL. This reduces TOL overhead of looking up a translation in the code cache.

The example binary translation/optimization mechanism that we just saw has two stages: interpretation and one translation phase. However, there are systems with more translation stages, with each translation stage applying progressively more complex optimizations. Moreover, there are some systems that skip interpretation stage and directly go to translation like IA-32 EL [Baraz et al. 2003] and DynamoRIO [Bruening et al. 2003]. The different interpretation/translation stages provide a tradeoff between startup and steady state performance. For example, applying aggressive optimizations is costly in terms of overhead; however, they generate a highly optimized code that runs faster than un-optimized code. Hence, a system that starts with aggressive optimizations, skipping interpretation and simple translation, would have unacceptably poor startup performance and excellent steady state performance. However, the overall performance of such a system would depend on how much of the startup delay or translation/optimization overhead could be offset by the optimized code execution. They might end up having poor overall performance if the translation/optimization overhead in not compensated by the optimized code execution. Therefore, most systems start with interpretation or lightweight translations to improve startup performance, whereas aggressive optimizations are applied only to hot code that dictates the steady state performance.

2.2 Salient features of HW/SW Co-designed Processors

HW/SW co-designed processors provide certain features that set them apart from traditional hardware only processors. These features include:

Hardware Simplicity: These processors employ simple hardware to cut down the complexity. To simplify the hardware they implement a simple RISC ISA. Furthermore, TOL is implemented as a software layer whereas, the hardware implementation of TOL in conventional CISC processors contributes significantly to the hardware complexity.

Power Consumption: Having a simple hardware allows HW/SW co-designed processors to keep power consumption within limits. The simple RISC ISA allows to have a simple front-end and avoid power hungry components.

Flexibility: The software implementation of TOL makes it relatively simple to upgrade a processors by introducing new features in the software layer, in the field. On the other hand, due to the hardware implementation, the conventional CISC processors cannot introduce new features in TOL or fix a bug once the processor is rolled out.

Multiple Guest ISA Support: HW/SW co-designed processors also provide an excellent opportunity to run multiple guest ISA on a single host ISA. In this case, TOL needs to support multiple front-ends where each front-end corresponds to a different guest ISA. Once a front-end has done guest ISA to intermediate representation translation, the common back-end can be used to generate the host ISA code. This feature allows the codes compiled for different architectures to be executed on the same hardware. This is going to be especially important in the future architectures as one would like be able to run any application on any computing device.

Binary Compatibility: TOL also allows HW/SW co-designed processors to maintain forward and backward binary compatibility without any additional hardware complexity. TOL can translate binaries targeted for old architectures to run them on a latest one and vice-versa.

3. MOTIVATION

Traditional compile time loop vectorization is effective for applications involving explicit array accesses since memory dependence analysis are relatively easy. Significant performance gains have been reported using compiler vectorization in the past [Bik et al. 2002; Larsen et al. 2000]. However, one of the major obstacles in vectorization at compile time is memory disambiguation and dependence testing. J. Holewinski [Holewinski et al. 2012] showed that static vectorization fails to extract significant vectorization opportunities especially in pointer-based applications. Furthermore, S. Maleki [Maleki et al. 2011] showed that the modern compilers, including Intel ICC, IBM XLC, and GNU GCC, are limited in vectorizing modern applications. Extensive use of pointers and pointer arithmetic in these applications complicate memory disambiguation and dependence testing.

Compilers need to do accurate interprocedural pointer disambiguation and interprocedural array dependence analysis to guarantee the correctness of vectorized code. However, S. Maleki [Maleki et al. 2011] showed that they lack this ability and hence, loose significant vectorization opportunities. We propose a speculative dynamic vectorization technique that relies on the fact that a pair of memory accesses rarely alias until and unless aliasing is obvious [Guo et al. 2006]. By speculatively assuming that two memory reference will never alias, unless aliasing is obvious, we avoid the need of accurate interprocedural pointer disambiguation and interprocedural array dependence analysis. As a result, the speculative vectorization catches the vectorization opportunities missed by the compilers.

Figure 3a shows a function with a loop that performs pointer arithmetic. The function takes in two pointers as parameters. Since the function can be called from a number of places in the entire program, the inter-procedural analysis of compiler needs to check whether the two pointers can alias or not. If the compiler cannot prove that the two pointers always reference different memory locations, it will conservatively assume dependence between them to ensure correctness.

As stated before, another approach to vectorization, SLP [Larsen et al. 2000], performs vectorization at lower intermediate representation level. SLP vectorizes at

basic block level instead of loop level. Therefore, SLP may vectorize portions of a loop if the whole loop is not vectorizable, whereas traditional loop vectorizers vectorize either whole loop or nothing. SLP starts by identifying adjacent memory accesses and then follows their def-use and use-def chains. Figure 3b shows low level intermediate representation for the loop of Figure 3a after unrolling it once. In this case, even though *I0* and *I6* are adjacent memory references, they cannot be packed by SLP since *I4* and *I6* may alias. Similarly, *I4* and *I10* access consecutive memory locations. However, they also cannot be vectorized because *I6* might alias with them. Thus, memory dependences affect both traditional loop vectorizers as well as SLP.

```
void example(double *a, double *b)
{
    int i;
    for (i = 0; i < NUM_ITR; i++)
        a[i] += b[i] * CONST;
}
```

a) An example loop with pointers.

```
loop:   I0    ld_64       v2, M [r2 + r1 * 8]
        I1    mulsd       v3, v2, v1
        I2    ld_64       v4, M [r3 + r1 * 8]
        I3    addsd       v5, v4, v3
        I4    st_64       v5, M [r3 + r1 * 8]
        I5    add         r4, r1, 1
        I6    ld_64       v6, M [r2 + r4 * 8]
        I7    mulsd       v7, v6, v1
        I8    ld_64       v8, M [r3 + r4 * 8]
        I9    addsd       xmm0, v8, v7
        I10   st_64       xmm0, M [r3 + r4 * 8]
        I11   add         r1, r4, 1
        I12   cmp         r1, r0
        I13   jne         loop
```

b) Unrolled lower level representation.

```
        V0    Pack              v1, v1, v1
loop:   V1    ld_128_spec       v2, M [r2 + r1 * 8]
        V2    mulpd             v3, v2, v1
        V3    ld_128            v4, M [r3 + r1 * 8]
        V4    addpd             v5, v4, v3
        V5    st_128_spec v5, M [r3 + r1 * 8]
        V6    add               r1, r1, 2
        V7    cmp               r1, r0
        V8    jne               loop
        V9    Unpack            xmm0, v5
```

c) Speculatively vectorized version.

Figure 3 An example loop with pointer arithmetic.

One possible solution that compilers may provide is to generate two versions of the loop: one without vectorization and another vectorized with a runtime test to check for aliasing. However, this solution is not optimal because:

(1) runtime test has to be executed every time before executing the loop, thus resulting in a performance loss. Moreover, as the number of arrays to be checked for aliasing increases the number of checks to be performed also increases.
(2) Having multiple versions of the loop increases the static code footprint of the application, which results in higher instruction cache size requirements.

Another way of vectorizing the example loop is through "__restrict" keyword. It can be used to indicate that a symbol is not aliased in the current scope. If the programmer knows that the two pointers to the function will not alias in any case, this information can be passed to the compiler using the "__restrict" keyword. Once the compiler is sure that the two pointers always access non-overlapping memory locations, it can vectorize the loop. However, this solution requires source code modification which is not always possible e.g. unavailability of the source code or any other reason. In contrast, the proposed mechanism does not require any source code modification.

HW/SW Co-designed processors provide an excellent opportunity to handle these cases: instead of generating multiple versions, a single speculatively vectorized version can be generated by the software layer and the hardware can be tailored to execute the vectorized code efficiently and safely. The proposed algorithm speculatively reorders memory operations to expose vectorization opportunities. For example, in the code of Figure 3b, our algorithm speculatively assumes that *I4* and *I6* will never alias and reorders them to pack *I0* and *I6* together, as shown in Figure 3c. Moreover, due to the speculative reordering, *V1* is converted to a speculative load and *V5* to a speculative store. If during the execution it turns out that *V1* and *V5* access overlapping memory locations, the hardware will detect this condition and will take corrective measures. In this example, by vectorizing speculatively, we are able to vectorize the whole loop, whereas loop vectorization and SLP could not find vectorization opportunities.

Therefore, having two complementary vectorizing schemes helps to get the best of both the worlds. First, static vectorization applies more complex and time consuming loop transformations, even though vectorizes conservatively. Later at runtime, a dynamic vectorizer catches the opportunities missed by static vectorization and speculatively vectorizes ambiguous memory references and their dependent operations.

4. VECTORIZATION ALGORITHM

This section provides the details of the proposed speculative dynamic vectorization scheme. Before explaining the vectorization algorithm itself, first we explain binary translation/optimization steps of the modelled HW/SW co-designed processor. It helps us understand the context in which vectorization is done.

The software layer of our co-designed processor is called Translation Optimization Layer (TOL). TOL operates in three translation modes for generating host code from the guest x86 code: Interpretation Mode (IM), Basic Block Translation Mode (BBM) and Superblock Translation Mode (SBM). SBM is the most aggressive translation/optimization mode and majority (more than 90%) of the dynamic application code is executed in this mode. Vectorization is done only in SBM, after applying several standard optimizations.

4.1 Pre-Vectorization Steps

Before starting with vectorization we create a superblock, apply standard optimizations on the superblock and generate a Data Dependence Graph (DDG). Each of these steps is explained below:

4.1.1 Superblock Creation

TOL starts by interpreting guest x86 instruction stream in IM. When a basic block is executed more than a predetermined number of times, TOL switches to BBM. In this mode, the whole basic block is translated and stored in the code cache and the rest of the executions of this basic block are done from the code cache. Moreover, profiling information is gathered for all the basic blocks in BBM using software counters. This information consists of execution and edge counters. The execution counters provide the execution frequency of basic blocks while the edge counters monitor the biased branch directions. Once the execution of a basic block exceeds another predetermined threshold, TOL creates a bigger optimization region, called superblock, using the branch profiling information collected during BBM.

In Superblock translation and optimization mode (SBM), TOL generates a new superblock starting from the triggering basic block. A superblock generally includes multiple basic blocks following the biased direction of branches. A superblock ends at one of the following conditions:

(1) The last basic block included in the superblock ends with an indirect branch, call, or return instruction.
(2) The last basic block included in the superblock ends with an unbiased branch or the probability of reaching the last basic block from the beginning of the superblock falls below a predetermined threshold.
(3) The number of instructions in the superblock exceeds a predetermined threshold.
(4) The number of basic blocks included in the superblock exceeds a predetermined threshold.

Moreover, the branches inside the superblocks are converted to “asserts” so that a superblock can be treated as a single-entry, single-exit sequence of instructions. This gives the freedom to reorder and optimize instructions across multiple basic blocks. “Asserts” are similar to branches in the sense that both checks a condition. Branches determine the next instruction to be executed based on the condition; however, asserts have no such effect. If the condition is true, assert does nothing. However, if the condition evaluates to false, the assert “fails” and the execution is restarted from a previously saved checkpoint in IM. Furthermore, if the number of assert failures in a superblock exceeds a predetermined limit, the superblock is recreated without converting branches to “asserts”. As a result, this time the superblock has to be treated as a single-entry multiple-exit sequence of instructions. Having multiple exits in a superblock also reduces available optimization opportunities because the instructions across different exit paths cannot be reordered as freely as before.

Furthermore, while creating a superblock, if a loop is detected, it is unrolled. Currently, we unroll loops consisting of only a single basic block, as they are the ones which provide maximum benefit [Muchnick 1997]. To detect and unroll the loops without control flow the following steps are followed.

(1) The target address of the first branch instruction in the superblock is compared against the address of the first instruction of the superblock. In case of a loop, the addresses match.
(2) The execution and edge counters are used to determine the loop trip count.
(3) Loop unroll factor is determined based upon the data types in the loop, SIMD accelerator width, and the loop trip count determined in the last step. For example, if a loop contains only single-precision floating-point data types, then for a 128-bit wide SIMD accelerator the loop is unrolled 4 times if the loop trip count is more than or equals to 4.

Moreover, the unrolled version of the loop is followed by the original loop (without unrolling). During execution, a runtime check is performed to determine whether to execute the unrolled version or the original loop. If the number of iterations left for execution are less than the loop unroll factor, then the original loop is executed instead of the unrolled loop.

4.1.2 Pre-optimizations

The optimizer applies several transformations on the superblock. First of all, x86 code is translated to an intermediate representation. Then the resulting code is transformed into a Static Single Assignment format. This transformation removes anti & output dependences and significantly reduces the complexity of subsequent optimizations. Second, a forward pass applies a set of conventional single pass optimizations: constant folding, constant propagation, copy propagation, and common subexpression elimination. Third, a backward pass applies dead code elimination.

After the basic optimizations, the Data Dependence Graph (DDG) is prepared. To create DDG, the input and output registers of the instructions are inspected and the corresponding dependences are added. During DDG creation, we perform memory disambiguation analysis. If the analysis cannot prove that a pair of memory operations will never/always alias, it is marked as "may alias". In case of reordering, the original memory instructions are converted to speculative memory operations. Apart from this, Redundant Load Elimination and Store Forwarding are also applied during DDG phase so that redundant memory operations are removed before vectorization. The DDG is then passed as input to the vectorizer. After vectorization, an instruction scheduler that uses a conventional list scheduling algorithm schedules the vectorized code. Afterwards, the determined schedule is used by the register allocator that implements linear scan register allocation algorithm. Finally, the optimized code is translated to the host instructions and is stored in the code cache.

4.2 The Vectorizer

This section explains the vectorization algorithm with pseudo-code and using a practical example. The pseudo-code for the vectorizer is listed in Algorithm 1. The vectorizer packs together a number of independent scalar instructions that perform the same operation, and replaces them with one vector instruction. The number of scalar instructions packed depends on two factors:

- data-types of scalar instructions
- host vector length

For example, for a host vector length of 128-bit, four 32-bit single-precision floating-point instructions can be packed together in a single vector instruction. Therefore, vectorization reduces dynamic instruction count and improves

performance. Before describing the algorithm itself, we define a set of conditions that a pair of instructions must satisfy to be included in the same pack:

- The instructions must perform the same operation.
- The instructions must be independent.
- The instructions must not be in another pack.
- If the instructions are load/store, they must be accessing consecutive memory locations.

Vectorization starts by marking all the instructions which are candidates for vectorization. Moreover, we mark *First Load* and *First Store* instructions. *First Load/Store* instructions are those for which there are no other loads/stores from/to adjacently previous memory locations. For example, if there is a 64-bit load instruction I_L that loads from a memory location [M] and there is no 64-bit load instruction that loads from address [M – 8], we call I_L *First Load*.

Vectorization begins by packing consecutive stores, starting from a *First Store*. The decision of starting with stores instead of loads is based on the observation that a given kind of operation always has the same number of predecessors, e.g. all the additions always have two predecessors, whereas the number of successors may vary depending on how many instructions consume the result. Consequently, following a bottom-up approach results in a more structured tree traversal than a top-down approach.

Once a pack of stores is created, their predecessors are packed (*Pack_pred_succ* rountine), before packing other stores, if they satisfy the packing conditions. Moreover, if the last store in the pack has a next adjacent store, it is marked as *First Store* so that a new pack can start from it.

Once all the stores are packed and their predecessor/successors chains have been followed, we check for remaining load instructions that satisfy the packing conditions and pack them in the same way as stores. *Pack_ldst* routine provides the functionality for packing loads and stores.

Vectorization starting from adjacent loads/stores has an obvious limitation: if a superblock does not have any consecutive loads/stores, nothing can be vectorized. To tackle this problem, after packing all loads/stores and their predecessors/successors, we check if still there are some arithmetic instructions that can be packed together. If yes, we vectorize them and follow their predecessor/successor trees *(Pack_Arith)*. This allows to partially vectorize loops with interleaved memory accesses.

While traversing the predecessor/successor chains, if we find out that the predecessors of a pack cannot be vectorized, a *Pack* instruction is generated. This *Pack* instruction collects the results of all the predecessors into a single vector register and feeds the current pack. Similarly, if all the successors of a pack cannot be vectorized, an *Unpack* instruction is generated. This *Unpack* instruction distributes the result of the pack to the scalar successor instructions. *Traverse_pred_succ* routine provides this functionality. For example, in the case of loops with interleaved memory access, when we reach several load instructions while traversing the tree, we find out that they cannot be packed since they are not consecutive. Therefore, we leave them in scalar form and assemble their results using a *Pack* instruction.

Moreover, *Pack* instructions are needed if a pack contains an instruction whose input is live-in of the superblock. Similarly, *Unpack* instructions are needed to put the results from a pack to the architectural registers that are live-outs of the superblock.

ALGORITHM 1A. TOP LEVEL VECTORIZATION FUNCTION

```
Vectorize (SB):
    Set_packable(SB,Available_for_pack, First_St,First_Ld)
    Pack_ldst(SB, Available_for_pack, First_St, packs)
    Pack_ldst(SB, Available_for_pack, First_Ld, packs)
    Set_Arith(SB, Available_for_pack, Arith)
    Pack_Arith(SB, Available_for_pack, Arith, packs)
```

ALGORITHM 1B. LOAD-STORE VECTORIZATION

```
Pack_ldst(SB, Available_for_pack, First_LdSt, packs):
    for inst in First_LdSt:
      vec_length = get_vector_length(inst)
      P = [inst]
      for i in range(1, vec_length):
        if inst has next_ldst:
          if inst_can_pack(P,next_ldst, Available_for_pack):
            P.extend(next_ldst)
            inst = inst.next_ldst
          else:
            break

      if len(P) == vec_length:
        packs.extend(P)
        Make_unavilable(P, Available_for_pack)
        First_LdSt.extend(inst.next_ldst)
        Traverse_pred_succ (SB, Available_for_pack, packs)
```

ALGORITHM 1C. VECTORIZE ARITHMETIC OPERATIONS

```
Pack_Arith(SB, Available_for_pack, Arith, packs):
    for inst in Arith:
      if inst in Available_for_pack:
        vec_length = get_vector_length(inst)
        P = [inst]
        for inst1 in Arith[pos(inst):len(Arith)]:
          if inst_can_pack(P, inst1, Available_for_pack):
            P.extend(inst1)
            if len(P) == vec_length:
              packs.extend(P)
              Make_unavilable(P, Available_for_pack)
              Traverse_pred_succ (SB, Available_for_pack, packs)
              break
```

ALGORITHM 1D. TRAVERSE PREDECESSORS/SUCCESSORS

```
Traverse_pred_succ(SB, Available_for_pack, packs):
    need_Pack = Pack_pred_succ(SB, Available_for_pack, packs[latest].preds, packs)
    if need_Pack:
      generate_Pack_inst
    need_Unpack = Pack_pred_succ(SB, Available_for_pack, packs[latest].succs, packs)
    if need_Unpack:
      generate_Unpack_inst
```

ALGORITHM 1D. VECTORIZE PREDECESSORS/SUCCESSORS

```
Pack_pred_succ(SB, Available_for_pack, pred_succ, packs):
    for inst in pred_succ:
      if inst in Available_for_pack:
        vec_length = get_vector_length(inst)
        P = [inst]
        for i in range(1, vec_length):
          for inst1 in pred_succ[i]:
            if inst_can_pack(P, inst1, Available_for_pack):
              P.extend(inst1)
              break
        if len(P) == vec_length:
          packs.extend(P)
          Make_unavilable(P, Available_for_pack)
          Traverse_pred_succ (SB, Available_for_pack, packs)
    if All_pred_succ_packed(pred_succ):
      return NO
    else
      return YES
```

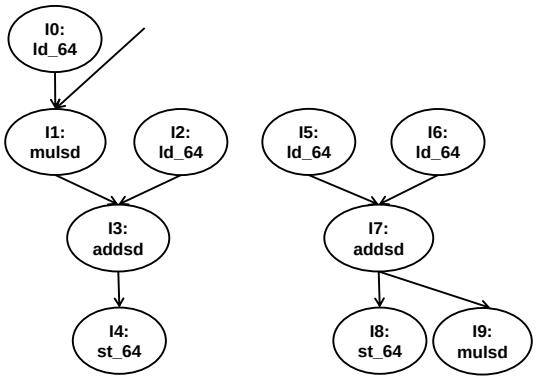
4.3 Avoiding Cyclic Dependences

One of the important points that should be taken care of during vectorization is that, after creation of a pack, two instructions that were earlier independent may become dependent. If we pack these instructions in a new pack, there will be a cyclic dependence in the DDG. Figure 4 shows an example of this scenario. Figure 4a shows the unvectorized code. We start vectorization by packing two consecutive and independent store instructions (*I4* and *I8*). Then following the predecessor chains we pack *I3* and *I7* also. After this step *I9* becomes dependent on *I1* as shown in Figure 4b, however these two instructions were independent in the original scalar code of Figure 4a. Therefore, we cannot select them to be packed together because it would produce a cyclic dependence.

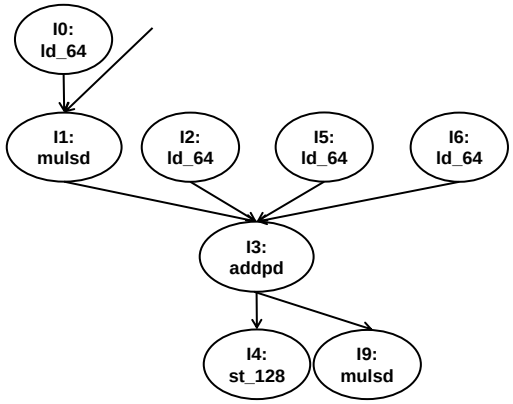
One way to solve the problem of inadvertently packing dependent instructions together is to address it during instruction scheduling and undo one of the packs involved in the cyclic dependence. However, it is not an optimal solution since dependence violation may have gotten propagated while traversing predecessor/successor chains. Therefore, we decided to update the DDG every time we create a new pack. As a result, cyclic dependences never appear in the DDG. This also allows us to check for alternative packing possibilities whereas, if we remove cyclic dependence during instruction scheduling, we cannot pack instructions of dissolved packs with other instructions.

4.4 Static vs Dynamic Vectorization

Loops are the basic program structures that the vectorizers target for extracting parallelism through vectorization. Several loop transformations are sometimes needed to make a loop vectorizable. The transformation like loop distribution, loop interchange, loop peeling, node splitting, memory layout change, algorithm substitution, etc are generally applied to make a loop vectorizable. These time consuming transformations are better suited at compile time than at runtime and therefore, these are not included in the proposed speculative dynamic vectorizer.



a) DDG for unvectorized code.



b) DDG after vectorizing I4-I8 and I3-I7.

Figure 4. Additional dependence after vectorization.

However, compile time vectorization suffers from several limitation like: 1) limited vectorization opportunities due to compilers inability to do accurate interprocedural pointer disambiguation and interprocedural array dependence analysis, 2) scope of vectorization is limited to basic blocks if the loops cannot be unrolled e.g. due to complex control flow, and 3) legacy code cannot be vectorized.

The proposed speculative dynamic vectorization gets rid of all these limitations. 1) The proposed algorithm avoids the need for accurate interprocedural pointer disambiguation and interprocedural array dependence analysis by speculatively assuming that ambiguous memory reference are independent, unless dependence is obvious, 2) Since the scope of vectorization for the proposed algorithm is a superblock, it crosses the basic block boundaries to vectorize instructions from multiple basic blocks along the most frequently executed paths, and 3) Since the dynamic vectorization is applied at runtime on the program binary and not at the source code level, the legacy code can also be vectorized.

Moreover, dynamic vectorization provides some additional benefits. For example, for the loops where the number of iterations are not known statically, it is difficult to decide the unroll factor at compile time. The availability of dynamic application behavior, at runtime, allows to detect the loop unroll factor dynamically. Unrolling the loops correspondingly helps dynamic vectorizer to extract significant vectorization opportunities.

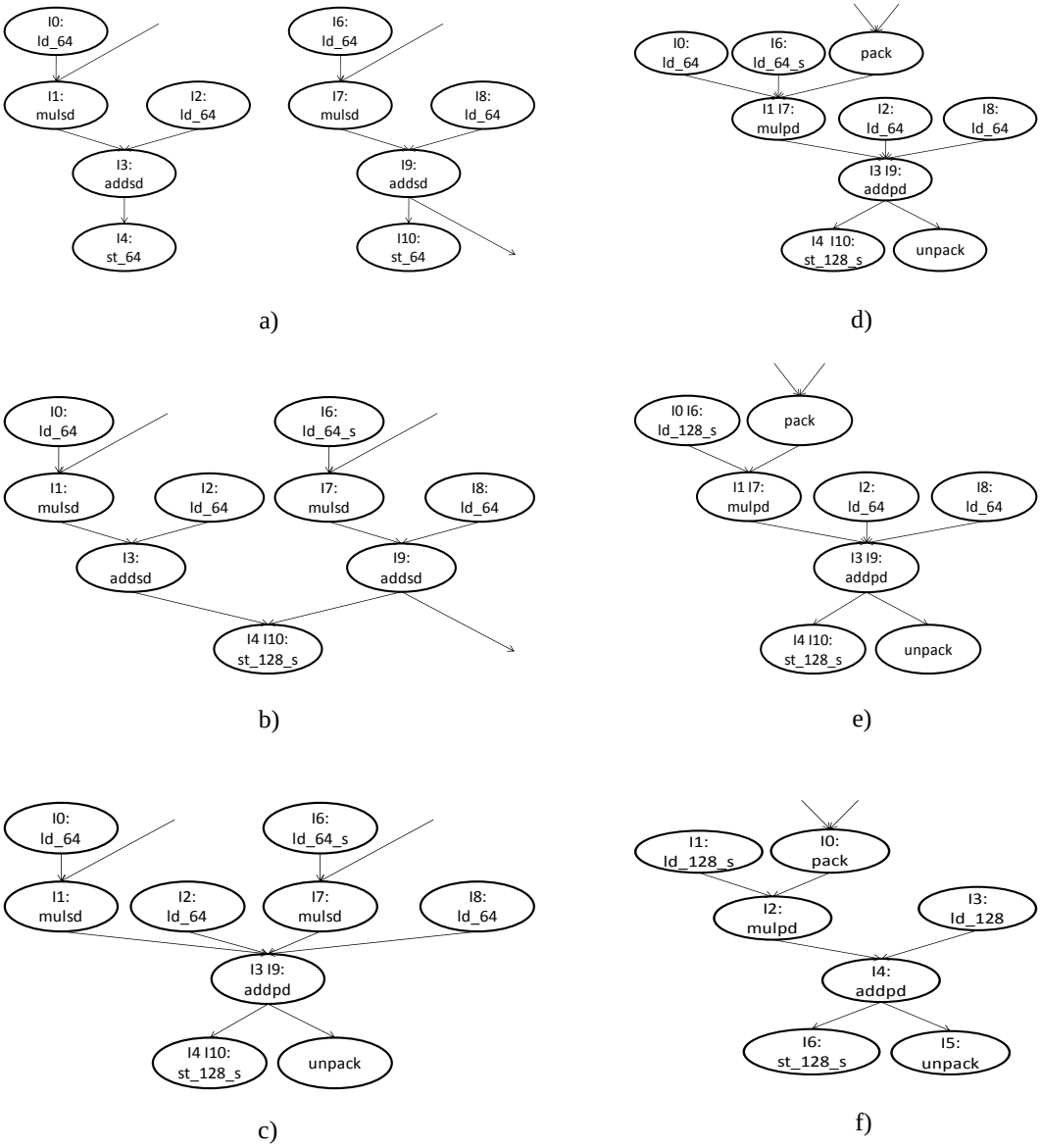


Figure 5 Example for vectorization of the code of Figure 3b. a). Shows the DDG for the loop which is unrolled once. We don´t show loop control code for the sake of simplicity. Since two iterations are completely independent we have two completely separated trees. Two arrows coming in to I1 and I7 represents live-in and arrow going out of I9 represents live-out of the superblock. Also, speculatively, we assume there is no dependence between the memory instructions until and unless its obvious b) Shows the state of DDG after vectorizing consecutive stores, also, the new store instruction is speculative one. c) Then, we follow the predecessor chains and pack addsd instructions. Since I9 writes to an architectural register, we need to unpack the results and write to the architectural register. d) Packs two mulsd instructions and since one of the inputs to both of these instructions is a live-in, a Pack instruction is also generated to pack the inputs. e) and f) pack remaining load instructions and f) Shows the final state.

4.5 Working through an Example

Figure 5a shows the DDG for the example code of Figure 3b. Since the loop is unrolled once and there is no loop carried dependences, assumed speculatively, the two trees are completely independent of each other. For the sake of simplicity, we do

not show loop control code in this figure. Also, the pairs of ambiguous memory reference instructions like *I4* and *I6* are considered independent speculatively. As our algorithm begins with consecutive stores, the stores *I4* and *I10* are packed together as shown in Figure 5b. Moreover, the new store instruction is speculative one and *I6* is also converted to speculative load. Following the predecessor tree, we see that *I3* and *I9* satisfy the packing conditions and vectorize them. Notice here that *I9* writes to a live-out architectural register. As a result, we have to generate an *Unpack* instruction to write the result to the live-out register. This is shown in Figure 5c.

Traversing up the tree, we vectorize multiplication instructions *I1* and *I7*. One of the inputs of the multiplication instructions is a live-in to the superblock. Hence, we generate a *Pack* instruction to put the live-in values in a vector register as shown in Figure 5d. As explained earlier, before packing the other predecessors of additions (*I3* and *I9*), we traverse the tree up for the predecessors of *I1* and *I7*. We discover that the loads *I0* and *I6* are independent and consecutive, thus, they are packed next. Also, the new vector load instruction is speculative since *I6* was speculative, Figure 5e. Finally, Figure 5f shows the second inputs of additions (*I3* and *I9*): the two load instructions (*I2* and *I8*) are also vectorized. *Pack* and *Unpack* instructions generated to read and write architectural registers in this example can be moved outside the loop as loop invariant code during instruction scheduling, as shown in Figure 3c. This way, we are able to vectorize the whole loop.

5. SPECULATION AND RECOVERY

Memory speculation is a key optimization to achieve performance in HW/SW co-designed systems. Considering two ambiguous memory references independent of each other provides more freedom in instruction scheduling and boosts performance. For example, Transmeta Crusoe [Dehnert et al. 2003] reports that, on average, suppressing memory reordering causes 10% and 33% performance loss in operating system boots and user applications respectively. Since, memory operations play an important role in vectorization, by freely reordering them consecutive memory references can be packed together. This not only helps in utilizing memory bandwidth but also in vectorization of their dependent arithmetic operations. Furthermore, it is important to note that HW/SW co-designed processors like Transmeta Crusoe, BOA etc provide hardware support for speculation and recovery even though they do not have any dynamic vectorization scheme. Therefore, we assume this hardware support to be present in our baseline architecture. Hence, from the vectorization point of view, we do not need to add any new hardware support for speculation and recovery. This section briefly explains how the speculation and recovery mechanism works in the modelled HW/SW co-designed processor.

A combination of software and hardware mechanisms is used to detect speculation failure and subsequent recovery. As described earlier, if a pair of memory references cannot be proved never/always aliasing; it is marked as “may alias”. TOL labels each load/store instruction with a sequence number in the original program order. If a pair of load-store or store-store instructions that may alias is reordered, the original load/store instructions are converted to “speculative load/store” instructions.

The hardware has two sets of architectural registers: a working set and a shadow copy. Before starting the execution of speculative code, a copy of the working set is saved into the shadow registers (saving a checkpoint). During the execution, only the working copy of the registers is updated. In the case of speculation failure,

Seq Num

```
1    ld_64     v1, M[x]
2    st_64     v2, M[y]
```

a) Original Code Sequence

Seq Num

```
2    st_64_s     v2, M[y]
1    ld_64_s     v1, M[x]
```

b) Reordered Code Sequence

PC --> `1  ld_64_s  v1, M[x]`

Seq Num	Address	Size
2	y	8

c) Hardware Table State

Figure 6 Speculation Failure Detection Example.

the register state is restored by copying the contents of shadow registers to the working copy. Restoring the memory state is a little more complicated since it is not practical to have two copies of the whole memory state. To track the changes in the memory state a store buffer is used. During the normal execution, store instructions write to the store buffer instead of directly writing to the memory. In the case of speculation failure, the contents of the store buffer are discarded, whereas they are forwarded to the memory if the speculated code executes successfully.

To detect a speculation failure, the hardware maintains a table to record address and size of all the memory locations accessed by "speculative load/store" instructions in the current superblock. Moreover, the sequence number of "speculative load/store" instructions is also recorded in the table. During the execution, if the hardware detects:

- that a speculative memory instruction with higher sequence number has been executed before another speculative memory instruction with lower sequence number and
- they access overlapping memory locations,

an exception is raised. In this case, the contents of the store buffer are flushed; register values from the shadow registers are copied to the working set; (this has the effect of restoring the earlier saved checkpoint) and the execution is restarted in Interpretation Mode. On the other hand, in case of successful execution of speculated code, values in the store buffer are forwarded to the memory and the contents of the shadow registers are discarded.

Figure 6 shows an example of speculation failure detection mechanism. Figure 6a shows the original code sequence with two memory references where the relation between the memory addresses is unknown. The two instructions are labeled in the program order. Figure 6b shows the reordered code sequence. The instructions maintain their sequence number. However, they are converted to speculative instructions to inform the hardware to check them for speculation failure. Figure 6c shows the hardware table state just before executing the speculative load instruction. The program counter points to the current instruction and the table has entry for the executed speculated store instruction. At this point, since the instruction with a higher sequence number (2) has been executed before the instruction with a smaller sequence number (1), if the address of the current speculated load instruction overlaps with the address of the speculated store instruction, the hardware will generate an exception and will go to the recovery mode.

If the rate of speculation failure exceeds a predetermined limit in a particular superblock, it is recreated without reordering ambiguous memory references. With this speculation and recovery support available in the baseline architecture, speculatively vectorized code can be executed correctly without any additional hardware support.

6. PERFORMANCE EVALUATION

6.1 Experimental Framework

To evaluate the proposals, we use DARCO [Pavlou et al. 2011], which is an infrastructure for evaluating HW/SW co-designed virtual machines. DARCO executes guest x86 binary on a PowerPC-like RISC host architecture. Since DARCO emulates floating point code in software, we extended the infrastructure to add floating point scalar and vector operations. The proposed algorithm was implemented in TOL to support vectorization.

In our experiments, we assume that the host architecture supports a vector width of 128-bits. Moreover, we consider only floating point operations for vectorization (because most SIMD optimizations tend to focus on them) and no integer operation is vectorized. For this reason, we show only floating point instructions in the results presented in this section.

For the speculation and recovery, as discussed in Section 5, the hardware maintains a table where it stores the sequence number, address and size of speculative load/store instructions. We implement this table with 1K entries. Optimal duration/position to take a checkpoint is a different research problem and is out of the scope of this paper. For simplicity we take checkpoint at the beginning of every superblock. We implement the store buffer with 1K entries. Moreover, to avoid overflow of the store buffer we restrict the number of load/store instructions to 1K in a superblock. Since we take checkpoint in the beginning of every superblock and a superblock cannot have more than 1K load/store, the store buffer can never overflow.

6.2 Benchmarks

To measure the success of the proposals we use a wide variety of benchmarks. First of all, we use TSVC (Test Suite for Vectorizing Compiler) [Maleki et al. 2011] benchmark suite to measure the effectiveness of speculative dynamic vectorization in vectorizing synthetic loops that Intel ICC failed to vectorize due to conservative memory disambiguation analysis. Secondly, a set of applications from SPECFP2006 [Standard Performance Evaluation Corporation] and Physicsbench [Yeh et al. 2007] benchmarks suites is used to measure the efficacy of our proposals in the real world applications. Furthermore, to measure the success of the proposed algorithm in vectorizing pointer based applications we use kernels from UTDSP benchmark suite [UTDSP Benchmarks]. UTDSP benchmark suite contains array and pointer based version of several signal processing kernels. Both versions provide identical functionality, the only difference being the use of arrays or pointers to traverse the data structures. SPECFP2006 benchmarks operate on double-precision, whereas Physicsbench and UTDSP operate on single-precision floating point values.

All the benchmarks are executed till completion. SPECFP2006 benchmarks are executed using the “train” input to keep the execution time manageable. We compare our results with both GNU GCC and Intel ICC compilers. The compiler versions and optimizations are listed in Table I.

Table I. Percentage of Dynamic Instructions eliminated by GCC, TOL and GCC+TOL vectorizations.

	GCC	ICC
Version	4.5.3	12.1.4
Baseline Optimization	-O3 -ffast-math -fomit-frame-pointer	-O3
Vectorization	-mfpmath=sse -msse3	-xSSE3
Disable vectorization	-fno-tree-vectorize	-no-vec

6.3 Test Suite for Vectorizing Compilers

The Test Suite for Vectorizing Compilers (TSVC) benchmark suite was developed by [Callahan et al. 1988] to assess the vectorization capabilities of compilers. The suite was originally written in Fortran. S. Maleki et al. [Maleki et al. 2011] translated it to C and also added additional loops to gauge the issues not addressed by the original suite. In our experiments we use the latter version.

To measure TOL vectorizer´s ability to catch the vectorization opportunities missed by static vectorization, we first find the loops that ICC could not vectorize. Then we feed these loops to TOL and check the vectorization results. ICC vectorization report details that 48 of the loops are not vectorized due to "`existence of vector dependence`". After passing through the TOL vectorization phase the vectorization status of these loops is as follows:

(1) Completely Vectorized loops: TOL vectorizer is able to completely vectorize 14 out of 48 ICC unvectorized loops. The loops that are completely vectorized by TOL are: s1113, s151, s162, s211, s1213, s1221, s241, s1244, s2251, s252, s261, s421, s422 and s424.

(2) Partially Vectorized Loops: 9 out of the remaining 34 loops are partially vectorized by the TOL vectorizer. These are the loops that are not completely vectorized however, more than 80% of the operations are vectorized. The main reason for not vectorizing the rest of the operations is noncontiguous memory accesses. TOL does not support indexed memory access nor gather-scatter, thus could not vectorize these memory accesses. The loops that fall under this category are: s212, s221, s222, s242, s243, s244, s281, s323 and s4114.

(3) Unvectorized loops: The rest of the loops are either partially vectorized (less than 80% of the operations) or not vectorized at all by the TOL vectorizer. The main reasons for not vectorizing these loops are: 1) Presence of control flow inside the innermost loop 2) Reductions, and 3) Irregular memory access patterns. The current version of TOL does not support any of these patterns.

As these results show, the speculative dynamic vectorization of TOL is able to completely vectorize around 30% of the loops that the static ICC vectorizer could not vectorize due to conservative memory disambiguation analysis. A further 18% of loops are partially vectorized. In total, 48% of loops are either completely or partially (more than 80% of the operations) vectorized by TOL vectorizer whereas Intel ICC vectorizer could not find any vectorization opportunities in these loops.

The next sections evaluate TOL vectorization using SPECFP2006, Physicsbench and UTDSP applications.

6.4 FP Dynamic Instruction Elimination

This section presents the percentage of dynamic instructions eliminated by 1) static compiler vectorization, 2) dynamic TOL vectorization and 3) static+dynamic vectorizations, first for SPECFP2006 and Physicsbench benchmarks suites and then for UTDSP Kernels. We present the results first using GCC as static vectorizer and

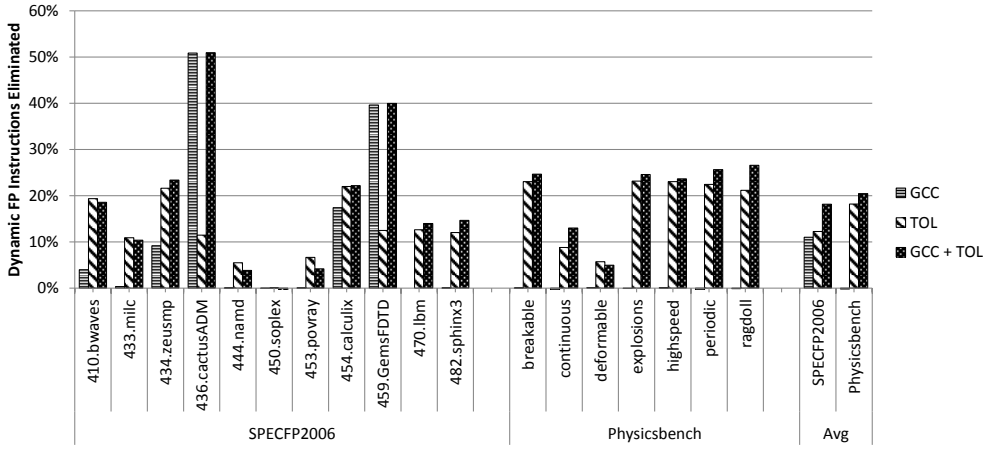


Figure 7 Percentage of Dynamic FP Instructions eliminated by GCC, TOL and GCC+TOL vectorizations.

then switching to ICC for static vectorization. Dynamic vectorization results show TOL´s effectiveness in vectorizing legacy code, since input binary is not vectorized for any SIMD accelerator. For static + dynamic vectorization case, the input binary to dynamic vectorizer (TOL) is already vectorized by the static vectorizer (GCC or ICC). The results of this case show the vectorization opportunities missed by GCC and ICC but captured by TOL.

6.4.1 Benchmarks

For SPECFP2006, on average, the combined GCC+TOL approach eliminates approximately twice the number of dynamic instructions than only the static GCC vectorization as shows in Figure 7. GCC+TOL vectorization outperforms GCC for all the SPECFP2006 benchmarks except for 436.cactusADM and 459.GemsFDTD. GCC completely vectorizes these benchmarks and hence TOL does not get any further vectorization opportunities. Therefore, instruction elimination is same for GCC and GCC+TOL. It is also important to note that on average, dynamic TOL vectorization itself slightly outperforms static GCC vectorization. Moreover, the only benchmarks where GCC outperforms TOL are again 436.cactusADM and 459.GemsFDTD. The effectiveness of TOL vectorization, to some extent, depends on the quality of the input binary. For example, for 436.cactusADM the input binary to TOL contains GCC unrolled version of the hottest loop. This GCC unrolled loop is split into multiple superblocks due to TOL´s restriction on the maximum number of instructions in a single superblock. Therefore, TOL vectorizer could not vectorize it as good as GCC. For 459.GemsFDTD, GCC generates significant spill-fill code in the frequently executed loops. This spill-fill code affects TOL´s ability to vectorize this benchmark.

GCC could not vectorize Physicsbench mainly due to the presence of complex control flow in the most frequently executed loops. TOL also is unable to unroll these loops; however, it extracts significant vectorization opportunities through superblock vectorization. Since GCC fails to vectorize anything, GCC+TOL and TOL vectorizations both eliminate 20% of the dynamic instruction stream.

As Figure 8 shows, for SPECFP2006 ICC+TOL vectorization outperforms the static ICC vectorization by eliminating 1.3x more instructions. Just like GCC+TOL vectorization, ICC+TOL always performs better or at least as good as ICC only vectorization. The benefit of ICC+TOL vectorization is especially evident in 459.GemsFDTD where the combined static + dynamic vectorization scheme

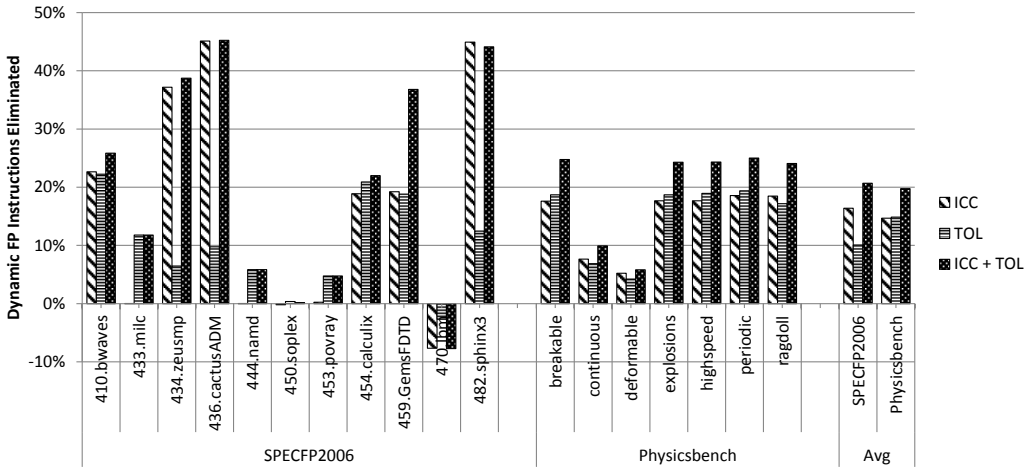


Figure 8 Percentage of Dynamic FP Instructions eliminated by ICC, TOL and ICC+TOL vectorizations.

eliminates twice the instructions compared to the static ICC approach. Moreover, there are benchmarks like 433.milc, 444.namd and 453.povray where ICC does not vectorize at all whereas, TOL and ICC+TOL vectorizations are able to find vectorization opportunities. On the other hand, 470.lbm suffers an instruction increase after vectorization. It is important to note that TOL vectorization is able to achieve dynamic instruction reduction for 470.lbm when it is compiled with GCC. However, for ICC compiled (unvectorized) version TOL vectorization also suffers instruction increment. As stated before, the quality of the TOL vectorization depends on the input binary. For example, the binary for a loop without any control flow may contain one or more basic blocks. If it has only one basic block, TOL can unroll and vectorize it however, for more than one basic blocks TOL will not unroll it.

Similarly, for Physicsbench ICC+TOL vectorization outperforms ICC only vectorization by 1.35x. For all the benchmarks in the Physicsbench TOL and ICC vectorizations perform equally well however, ICC+TOL catches additional vectorization opportunities at runtime.

Table II. Percentage of Dynamic Instructions eliminated by different vectorizations schemes.

Benchmark	Type	GCC	TOL (GCC in)	GCC + TOL	ICC	TOL (ICC in)	ICC + TOL
FFT	Array	43.28%	52.70%	43.28%	53.50%	49.98%	53.50%
	Pointer	0.00%	49.87%	49.87%	0.00%	49.98%	49.98%
FIR	Array	0.00%	0.00%	0.00%	7.96%	0.00%	7.96%
	Pointer	-0.08%	0.00%	-0.08%	0.00%	0.00%	0.00%
IIR	Array	0.00%	32.52%	32.52%	0.00%	31.39%	31.39%
	Pointer	0.00%	0.00%	0.00%	0.00%	-3.84%	-3.84%
LATNRM	Array	23.48%	7.38%	20.44%	21.75%	17.68%	29.21%
	Pointer	19.43%	17.85%	27.76%	19.80%	20.36%	30.77%
LMSFIR	Array	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	Pointer	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
MULT	Array	64.72%	17.62%	64.72%	70.63%	17.60%	70.63%
	Pointer	0.00%	17.62%	17.62%	40.73%	17.60%	40.73%
Avg	Array	21.91%	18.37%	26.83%	25.64%	19.44%	32.11%
	Pointer	3.23%	14.22%	15.86%	10.09%	14.02%	19.60%

6.4.2 Kernels

Table II shows the vectorization results for UTDSP kernels. As the table shows GCC vectorizes the array based version of FFT, LATNRM and Matrix Multiplication (MULT) but for the pointer based version it is able to vectorize only LATNRM. On the contrary, TOL is equally effective in vectorization of array and pointer based versions for all the kernels except for IIR. Pointer based version of IIR contains control flow inside the innermost loop and hence TOL fails to vectorize it. Furthermore, once again the combination of static and dynamic vectorization, GCC+TOL, provides the best solution.

For the array based version, TOL vectorizer outperforms GCC in vectorizing IIR. GCC is unable to resolve loop carried dependences, whereas speculative vectorization helps TOL to provide an instruction reduction of 32%. On the other hand, GCC surpasses TOL vectorization for LATNRM and Matrix Multiplication (MULT). In the current version of TOL vectorizer, reductions are not vectorized. Both LATNRM and MULT employ reductions, which TOL fails to vectorize. Moreover, MULT has non-unit stride memory accesses, since only one dimension of the matrix (either row or column) can be accessed in unit-stride manner. Compilers apply optimizations like "memory layout change", "data coping" etc to convert non-unit stride accesses to unit-stride. However, these optimizations are not directly applicable at runtime. This adds to the loss of vectorization opportunities for TOL vectorizer.

The vectorization results with ICC vectorization are similar to those of GCC except for the pointer based version of the MULT kernel. ICC is able to vectorize it but still does not do as good job as for the array based version. Also the combination of static + dynamic vectorizers outperforms individual static and dynamic vectorization for both array and pointer based code.

None of the vectorization schemes is able to extract benefit for FIR and LMSFIR, mainly because of the presence of control flow inside the innermost loop. Moreover, in these benchmarks, the number of independent instructions in the basic blocks (and even in superblocks) is not enough to enable vectorization. It is also interesting to note that TOL eliminates 53% of instructions from array version of FFT, whereas GCC+TOL eliminate only 43% (as does GCC alone). This is because the input to TOL is completely vectorized by GCC and TOL does not find any vectorization opportunities, therefore the instruction reductions stays at 43% in GCC+TOL case.

6.5 Dynamic FP Instruction Stream Distribution

Figure 9 and 10 present dynamic FP instruction stream distribution for SPECFP2006 and Physicsbench respectively for no vectorization, GCC vectorization, TOL vectorization, GCC+TOL vectorization and ICC+TOL cases. The results shown are normalized to no vectorization case. The dynamic FP instruction stream includes: Scalar and Vector instructions, *Pack/Unpack* instructions (as described in Section 4.2), unvectorizable instructions, and Merge instructions (the instructions needed to merge correct values in live-out architectural registers even without vectorization).

For GCC vectorization, the majority of the dynamic instruction stream is composed of scalar instructions. However, for TOL, GCC+TOL and ICC+TOL vectorizations the percentage of scalar instructions falls to 41%, 36% and 31% for SPECFP2006 and 57%, 50% and 52% for Physicsbench respectively. Furthermore, even though scalar instructions form much smaller (41%, 36% and 31%) part of the vectorized dynamic instruction stream in SPECFP2006 than Physicsbench (57%, 50% and 52%), the overall dynamic instruction stream for both benchmarks suites is reduced by the similar amount, almost 20%, by TOL, GCC+TOL and ICC+TOL

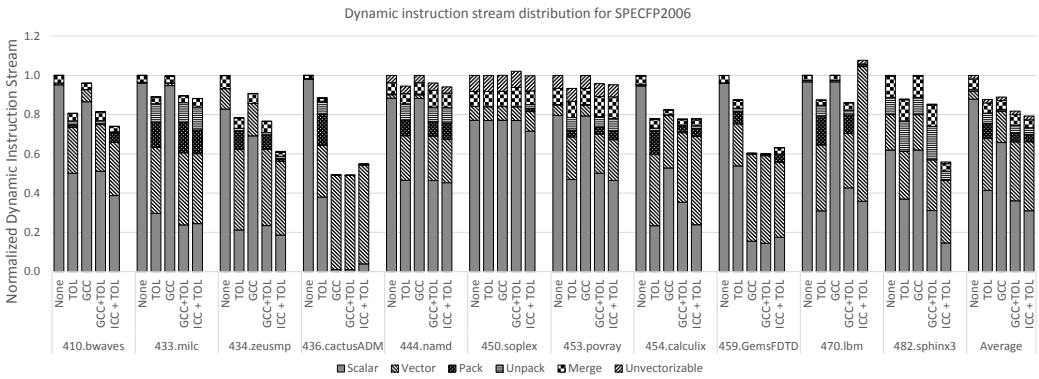


Figure 9 Dynamic FP instruction stream distribution for SPECFP2006: no vectorization, GCC, TOL, GCC+TOL and ICC+TOL vectorization normalized to no vectorization.

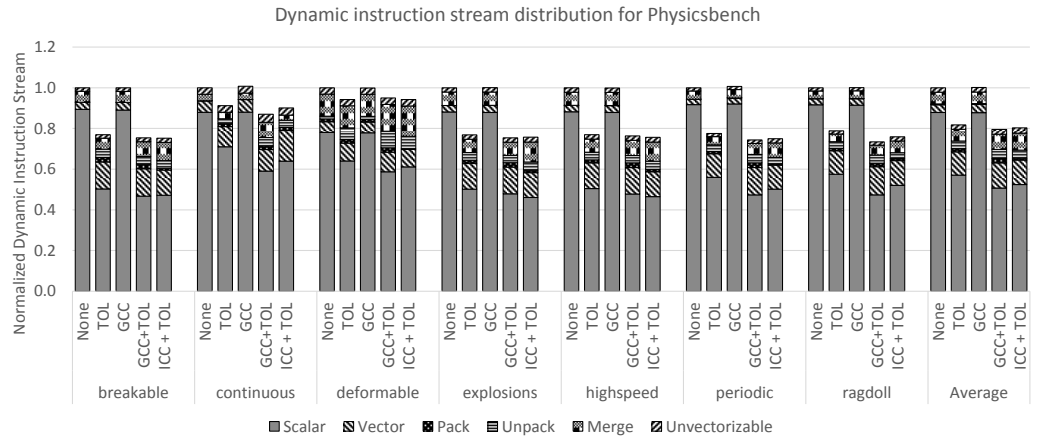


Figure 10 Dynamic FP instruction stream distribution for Physicsbench: no vectorization, GCC, TOL, GCC+TOL and ICC+TOL vectorization normalized to no vectorization.

vectorizations. The reason lies in the fact that SPECFP2006 benchmarks operate on 64-bit double-precision floating-point variables whereas, Physicsbench benchmarks are composed of 32-bit single-precision floating-point variables. As a result, for a vector length of 128-bits, a single vector instruction in Physicsbench replaces four scalar instructions whereas, in SPECFP2006 a vector instruction replaces only two scalar instruction. Therefore, SPECFP2006 needs more vector instructions to replace the same number of scalar instructions than Physicsbench. The fact is also evident in Figure 9 and 10 where the vector instructions form 26%, 30% and 35% of the vectorized instruction stream in SPECFP2006 and only 12% in Physicsbench for TOL, GCC+TOL and ICC+TOL vectorizations.

In addition, *Pack* and *Unpack* instructions also form a moderate fraction of the vectorized dynamic instructions stream. For TOL, GCC+TOL and ICC+TOL vectorizations, they constitute 13%, 8% and 7% of vectorized dynamic instruction stream for SPECFP2006 and 5% for Physicsbench. *Pack*/*Unpack* instructions are needed when the data needs to be reshuffled before it could be consumed by the following vector instructions, for example in complex pointwise vector multiplication. It is important to keep the number of *Pack*/*Unpack* instructions to a minimum, especially in wider vector units (256 bits and more), to avoid compromising the gains of vectorization. The problem of keeping *Pack*/*Unpack* instructions to a minimum is

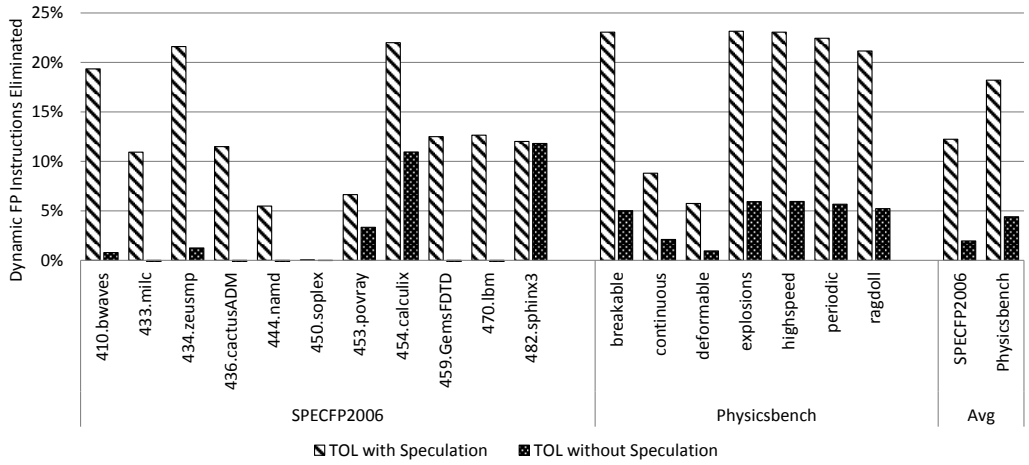


Figure 11 Percentage of Dynamic FP Instructions eliminated by TOL vectorizations with and without speculation.

orthogonal to the problem targeted in this paper and is discussed in detail in our other work [Kumar et al. 2013].

6.6 Importance of Memory Speculation

To understand the contribution and importance of memory speculation in dynamic vectorization we disabled the memory speculation while enabling TOL vectorization. Figure 11 shows the dynamic instruction eliminated for SPECFP2006 and Physicsbench respectively with and without memory speculation for TOL only vectorization. As the figure shows, disabling memory speculation results in severely limiting vectorization opportunities. With memory speculation TOL is able to reduce the dynamic instruction count by 12% and 18% for SPECFP2006 and Physicsbench respectively however, without memory speculation the dynamic instruction reduction is only 2% and 4% for these two benchmark suites. These results are worse than the static GCC vectorization as well.

```
void example()
{
    int i;
    double a[NUM_ELEM],b[NUM_ELEM],c[NUM_ELEM];
    // read arrays a and b
    for (i = 0; i < NUM_ELEM; i++)
        c[i] = a[i] + b[i];
}
```

a) Source code for a simple addition loop.

```
loop:   I0    ld_64     v2, M [r2 + r1 * 8]
        I1    ld_64     v3, M [r3 + r1 * 8]
        I2    addsd     v4, v2, v3
        I3    st_64     v4, M [r4 + r1 * 8]
        I4    addi      r1, r1, -1
        I5    cmp       r1, r0
        I6    jne       loop
```

b) Assembly code for the same addition loop.

Figure 12 An example addition loop at source and assembly code level.

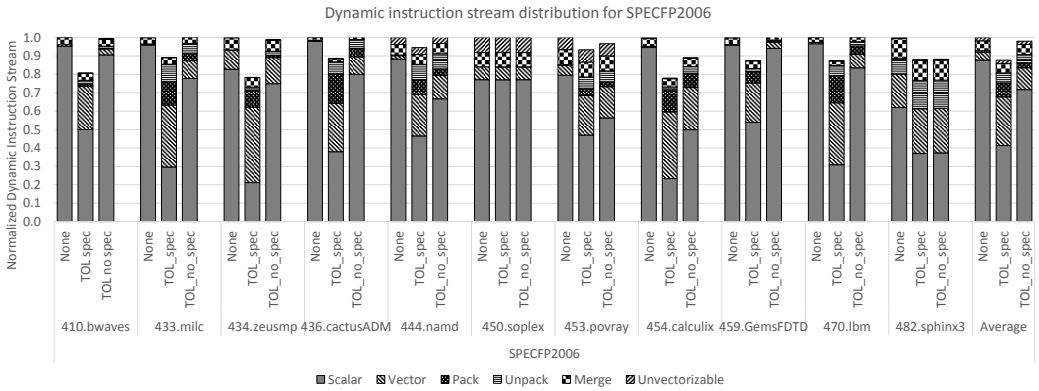


Figure 13 Dynamic FP instruction stream distribution for SPECFP2006: no vectorization, TOL with memory speculation (TOL_spec) and TOL without memory speculation (TOL_no_spec) vectorization normalized to no vectorization.

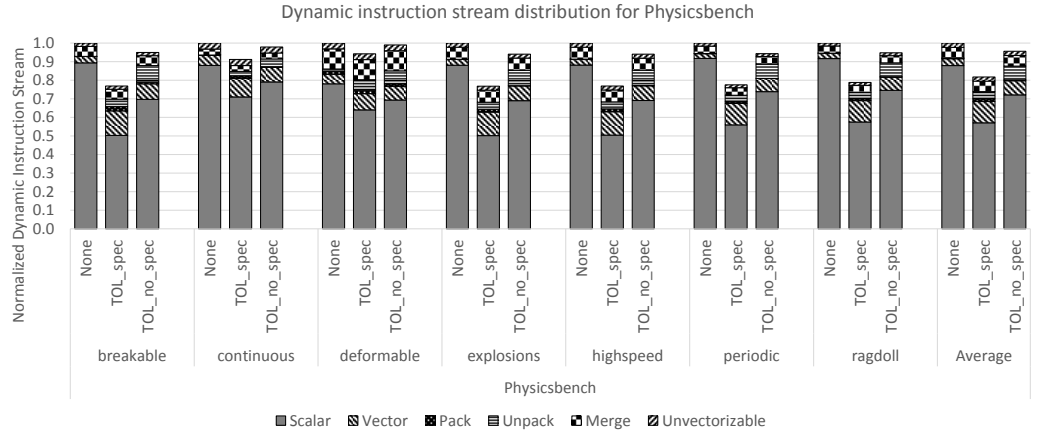


Figure 14 Dynamic FP instruction stream distribution for Physicsbench: no vectorization, TOL with memory speculation (TOL_spec) and TOL without memory speculation (TOL_no_spec) vectorization normalized to no vectorization.

The reason for having reduced vectorization opportunities without memory speculation lies in the fact that memory disambiguation is even more difficult at binary level than at the source code level. For example, in the source code of Figure 12a compiler can easily vectorize the loop since it adds two distinct arrays and saves the results in the third one. This information can be easily deduced at the source code level. On the other hand, in the binary code of Figure 12b, two registers r2 and r3 hold the base addresses of two input arrays a and b. The relation between the addresses held by these two registers is unknown. Therefore, it is not straightforward to determine whether the two registers hold the base addresses of two completely non-overlapping arrays or not. Hence, in the absence of memory speculation, the runtime vectorizer assumes that the arrays may overlap and does not vectorize the loop. This behavior drastically reduces runtime vectorizer's ability to extract vectorization opportunities. Therefore, memory speculation is not only a luxury but also a necessity for runtime vectorization.

Figure 13 and 14 shows the dynamic instruction distribution for SPECFP2006 and Physicsbench respectively with (TOL_spec) and without (TOL_no_sepc) memory speculation for TOL only vectorization. As the figures show, without memory

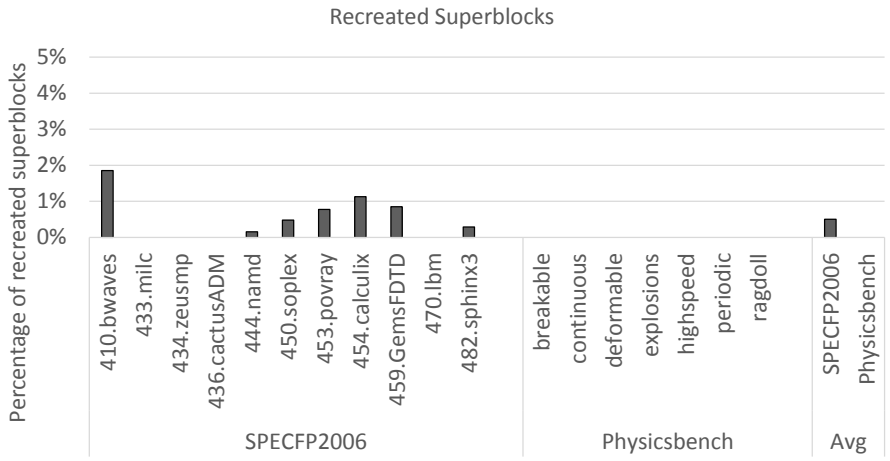


Figure 15 Percentage of static superblocks recreated due to memory speculation failure.

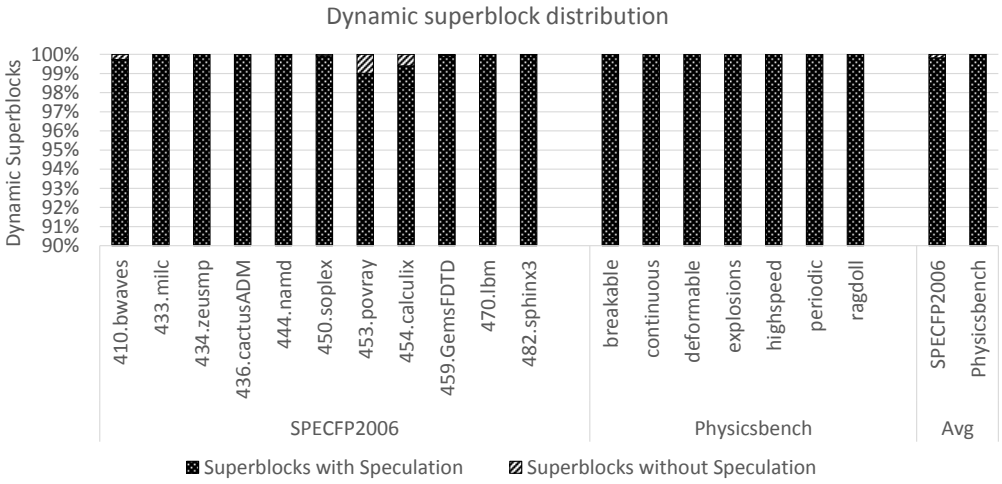


Figure 16 Dynamic superblocks executed with and without memory speculation.

speculation the vectorization coverage, the number of scalar instructions vectorized, drops drastically. The scalar instructions constitute only 40% (SPECFP2006) and 57% (Physicsbench) of the dynamic vectorized instruction stream with memory speculation enabled however, this number rises to 72% for both the benchmark suites when we disable memory speculation. These results show that the TOL vectorizer could not find much independent scalar instructions for vectorization in the absence of memory speculation. Therefore, a significant fraction of code is left unvectorized.

6.7 Robustness of Memory Speculation

One of the main factors in the success of the proposed dynamic vectorization scheme is the memory speculation. However, it might backfire if there are lots of speculation failures. A speculation failure results in executing un-optimized (and without TOL vectorization) version of the code and if the rate of speculation failure exceeds a predetermined threshold, recreating the superblock without speculation. Figure 15 shows the percentage of superblocks recreated due to memory speculation failure. As the figure shows, on average only 0.5% of superblocks are recreated in SPECFP2006

while in Physicsbench none of the superblocks had to be recreated. This shows that the rate of memory speculation failure is minimal.

The numbers shown in Figure 15 are for static superblock recreation. However, if one of the most frequently executed superblock is recreated without memory speculation, a significant amount of dynamic code might be executed without speculation. Figure 16 shows the percentage of dynamic superblocks executed with and without memory speculation. As the figure shows more than 99% of the dynamic code is executed with memory speculation. It reflects the fact that the number of speculation failures, and hence the overhead associated with it, is negligible.

The reason for not having noticeable speculation failures is the observation made by [Guo et al. 2006] that a pair of memory references rarely alias until and unless the aliasing is obvious.

6.8 Vectorization Overhead

Vectorization overhead is the fraction of dynamic instruction stream that corresponds to the vectorization of superblocks by TOL. A high vectorization overhead might offset the benefits of the vectorization. We calculate the vectorization overhead as:

$$= \frac{Total\ overhead_{with\ vectorization} - Total\ overhead_{without\ vectorization}}{Total\ number\ of\ dynamic\ instructions_{without\ vectorization}}$$

Our experimental results show that, on average, the vectorization overhead is less than 0.5% of the dynamic instruction stream, for all the benchmark suites. Hence, the dynamic vectorization overhead is negligible compared to its benefits. There are two main factors that make the vectorization overhead to be negligible:

First, since vectorization is performed at superblock level, the "superblock overhead" is the only overhead component that would increase. Moreover, the superblock creation overhead accounts for less than 10% of the overall overhead. Therefore, an increase in this component has minimal effect on the overall overhead.

Secondly, since vectorization reduces the total number of instructions in a superblock, the optimizations following the Vectorization pass, namely instruction scheduling, register allocation, and host code generation, now have to optimize lesser instructions. Therefore, the overhead of these optimization steps also reduces. As a result, total increase in the overall overhead is insignificant.

Table III. Processor Microarchitectural Parameters.

Parameter	Value
L1 I-cache	64KB, 4-way set associative, 64-byte line, 1 cycle hit, LRU
L1 D-cache	64KB, 4-way set associative, 64-byte line, 1 cycle hit, LRU
Unified L2 cache	512KB, 8-way set associative, 64-byte line, 6 cycle hit, LRU
Scalar Functional Units (latency)	2 simple int(1), 2 int mul/div (3/10) 2 simple FP(2), 2 FP mul/div (4/20)
Vector Functional Units (latency)	1 simple int(1), 1 int mul/div (3/10) 1 simple FP(2), 1 FP mul/div (4/20)
Registers	128-Integer, 128-Vector, 32-FP
Main memory Lat	128 Cycles

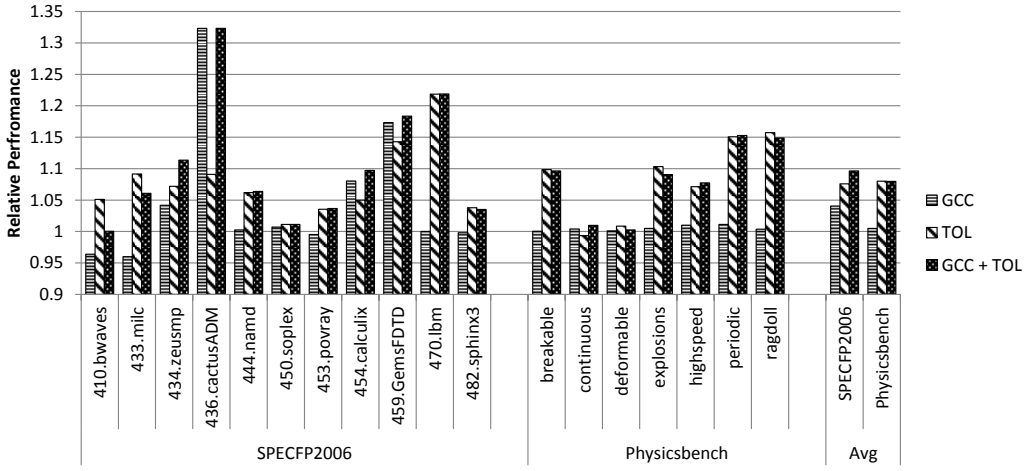


Figure 17 Execution speed for GCC, TOL and GCC + TOL vectorized code relative to unvectorized code. Higher is better.

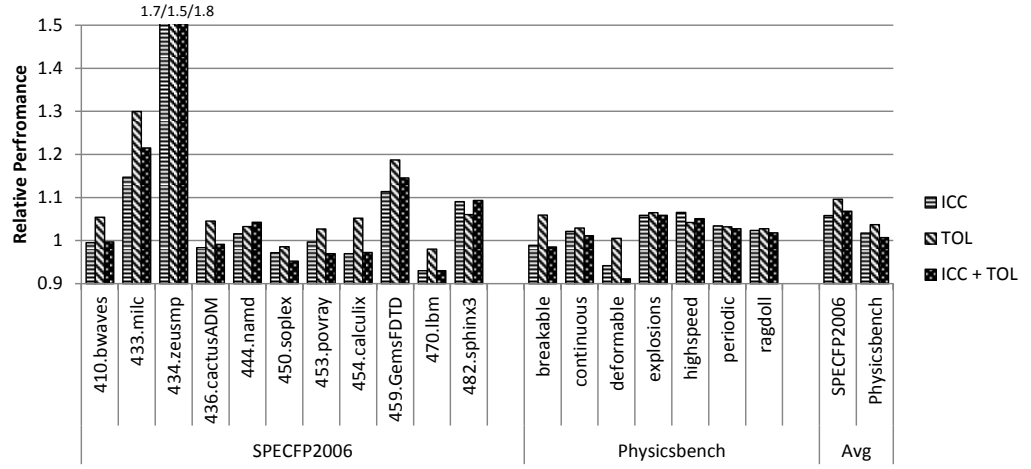


Figure 18 Execution speed for ICC, TOL and ICC + TOL vectorized code relative to unvectorized code. Higher is better.

6.9 Performance

For the performance analysis, we model a simple in-order processor, in congruence with the simple hardware design philosophy of the co-designed processors, with issue width of two. Microarchitectural parameters for the modeled processor are given in Table III. For the performance analysis both the floating point and integer code.

Figure 17 shows the performance of the vectorized code using the different vectorization schemes relative to the unvectorized code, for SPECFP2006 and Physicsbench. The performance results in the figure conform to the results of Figure 7 for dynamic instruction elimination. For SPECFP2006, GCC+TOL vectorization provides twice the performance benefit than GCC alone (10% compares to 5% of GCC alone). Also, TOL vectorization alone provides better performance than GCC alone. It is interesting to note that for 410.bwaves and 433.milc GCC vectorized code gets a slowdown even though Figure 7 shows dynamic FP instruction elimination. The slowdown comes because of the integer code. GCC adds more integer code than it

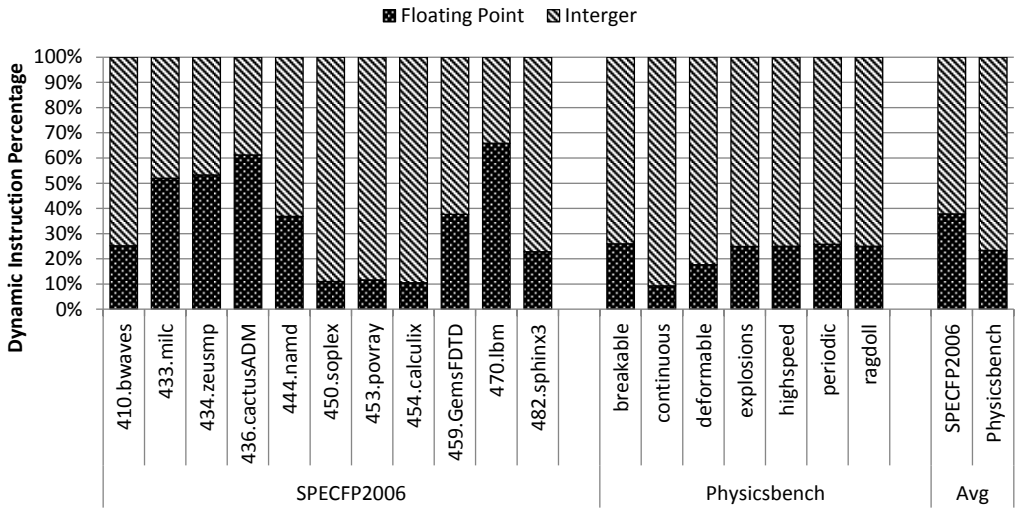


Figure 19 Integer and Floating Point Instruction Distribution in SPECFP2006 and Physicsbench

vectorizes, hence suffers a slowdown. Moreover, for these benchmarks GCC+TOL provides worse performance than TOL alone because GCC+TOL vectorizes GCC vectorized input with extra integer code whereas TOL vectorizes unvectorized code.

As GCC fails to vectorize anything in Physicsbench it does not show any performance improvements. However, similar to the results of Figure 7, GCC+TOL and TOL vectorizations provide similar performance benefits for Physicsbench.

An interesting thing to note is that in Figure 7 GCC+TOL vectorization, on average, eliminates approximately 20% of the dynamic instruction stream for both SPECFP2006 and Physicsbench. However, SPECFP2006 gets more speed up than Physicsbench as shown in Figure 17. This is because percentage of floating point code is more in SPECFP2006 than in Physicsbench as shown in Figure 19.

Figure 18 shows the performance results for ICC, TOL and ICC+TOL vectorization schemes. For SPECFP2006, ICC+TOL slightly outperforms ICC only vectorization as was the case for dynamic instruction reduction in Figure 8. The important point to notice in this figure is that TOL only vectorization outperforms both ICC and ICC+TOL vectorizations. This is due to the fact that ICC generates a number of checks in form of integer code to ensure the correctness of vectorized code. This additional integer code influences the overall performance of the application. Since TOL only vectorization does not add this code, it is able to achieve more performance over ICC vectorized code. Furthermore, since input binary for ICC+TOL already includes the additional integer code, ICC+TOL performance is lower than TOL only performance.

On the similar lines, TOL only vectorization outperforms both ICC and ICC+TOL for Physicsbench as well even though ICC+TOL have better instruction reduction as shown in Figure 8. The reason for this behavior is same as for the SPECFP2006 case.

Table IV shows the speedup for UTDSP kernels. These results also conform to the results of Table II. For the pointer based version of the kernels GCC loses significant performance compared to the array based version. However, performance is not affected a lot for TOL vectorizer. Furthermore, the combination of static and dynamic vectorizations, GCC+TOL, is able to extract maximum performance out of

the kernels. For the ICC vectorization also the results conform to the dynamic instruction reduction results of Table II for the majority of kernels.

Table IV Execution speed for code vectorized by different vectorization schemes relative to unvectorized code. Higher is better.

Benchmark	Type	GCC	TOL (GCC in)	GCC + TOL	ICC	TOL (ICC in)	ICC + TOL
FFT	Array	1.26	1.50	1.26	1.06	1.15	1.04
	Pointer	1.00	1.50	1.50	1.00	1.14	1.14
FIR	Array	1.00	1.00	1.00	1.74	1.04	1.72
	Pointer	1.05	1.00	1.05	1.00	0.98	0.98
IIR	Array	1.00	1.29	1.29	1.00	1.14	1.14
	Pointer	1.00	1.00	1.00	1.00	0.98	0.98
LATNRM	Array	1.39	1.03	1.33	1.37	1.04	1.33
	Pointer	1.31	1.13	1.39	1.32	1.04	1.31
LMSFIR	Array	1.00	1.00	1.00	1.00	0.99	0.99
	Pointer	1.03	1.00	1.03	1.00	0.98	0.98
MULT	Array	2.33	1.07	2.33	2.14	1.10	2.11
	Pointer	1.17	1.23	1.16	1.55	1.11	1.53
Avg	Array	1.33	1.15	1.37	1.39	1.08	1.39
	Pointer	1.09	1.14	1.19	1.15	1.04	1.15

7. REALTED WORK

Speculative Dynamic Vectorization is not a much extended topic in literature. There have only been a few proposals like Speculative Dynamic Vectorization [Pajuelo et al. 2002] and Dynamic Vectorization in Trace Processors [Vajapeyam et al. 1999]. None of them is in the context of HW/SW co-designed processors.

Pajuelo [Pajuelo et al. 2002] proposed to speculatively vectorize the dynamic instruction stream in the hardware for superscalar architectures. Their scheme prefetches data into the vector registers and speculatively manipulates it through arithmetic instructions. Moreover, scalar instructions that are converted into vectors are not eliminated but are converted into 'check' operations to validate whether the operands used by the corresponding vector instruction were correct or not. Several hardware structures are added to support speculative dynamic vectorization, which is not a power efficient solution, especially in out-of-order superscalar processors where power consumption is already a big issue. They report, more than half of the speculative work is unless due to mispredictions, whereas the rate of speculation failure is negligible in our case. S. Vajapeyam [Vajapeyam et al. 1999] builds a large logical instruction window and converts repetitive dynamic instructions from different iterations of a loop into vector form. The whole loop is vectorized if all iterations of the loop have the same control flow.

HW/SW Co-designed processors like Transmeta Crusoe [Dehnert et al. 2003], BOA [Sathaye et al. 1999], etc. apply several dynamic optimizations at runtime and evaluate their contribution in improving overall performance. Also, software dynamic binary optimizers like Dynamo [Bala et al. 2000], IA-32 [Baraz et al. 2003], and hardware dynamic binary optimizers like replay [Patel et al. 2001] and PARROT [Rosner et al. 2004] report performance improvements by applying on the fly optimizations. However, none of these systems have proposed vectorization at runtime. Y. Almog [Almog et al. 2004] briefly point out that one of the optimizations

applied in their system is SIMDification. Unfortunately, details of their vectorization scheme are not provided in the paper.

Traditionally, compiler vectorization targets loops for vector code generation. The vectorizer, first of all, strip-mines the loop iteration space by vector length. Then a vectorized version of the loop is generated along with some pre- and post-vectorization steps. This kind of loop vectorization operates at source code level and either whole loop is vectorized or nothing. S. Kral et al. [Kral et al. 2003] used FFTW, an automatic performance tuning system, to auto-vectorize FFT kernels and showed that the auto-vectorization can provide comparable performance to hand vectorized code. In contrast to traditional loop vectorization, SLP [Larson et al. 2000] vectorizes at low intermediate code level. This technique transforms loop level parallelism into superword level parallelism by unrolling the loop. Moreover, fractions of a loop can be vectorized if the whole loop is not vectorizable. J. Shin et al. [Shin et al. 2005] extended SLP in the presence of control flow. The basic idea behind their technique is to execute both if and else parts of an "if statement" in vector form and then choose the correct result based on the outcome of the control instruction.

Liquid SIMD [Clark et al. 2007] decouples the SIMD accelerator implementation from the instruction set of the processor by compiler support and a hardware based dynamic translator. Similarly, Vapor SIMD [Nuzman et al. 2011] provides a just-in-time compilation solution for targeting different SIMD architectures. Thus, both solutions eliminate the problem of binary compatibility and software migration. However, both need compiler changes and recompilation. J. Li [Li et al. 2006] propose a runtime algorithm for mapping guest vector registers to host vector registers when guest ISA vector registers support more data types than host ISA vector registers.

There has also been vectorization work in Java [El-Shobaky et al. 2009] [Nie et al. 2010]. S. El-Shobaky et al. [El-Shobaky et al. 2009] implement their vectorization technique in Jikes RVM to vectorize Java code. Their algorithm comprises of unrolling the loops, finding isomorphic instructions and replacing them with their vector counterpart. J. Nie et al. [Nie et al. 2010] present two vectorization approaches for Jitrino. The first approach is a library-based programming approach. For this one they define a generic set of Java vectorization interface with Java class library. The vectorized library functions can be used for vector code generation. The second approach is automatic vectorization in a Java virtual machine that does not require programmer assistance. They implement a loop-based vectorization with two phases. The first phase analyses and collects necessary information about the loop and the second phase transforms and vectorizes the loops. But none of these approaches use any kind of speculation to get additional vectorization opportunities as does our approach.

Previous work has also investigated improving the vectorization capabilities of compilers by making the underlying SIMD accelerator more flexible. V. Govindaraju et al. [Govindaraju et al. 2013] use a Coarse-grained Reconfigurable Architecture (CGRA) called DySER [Govindaraju et al. 2011] instead of a conventional SIMD accelerator. DySER consists of a configurable datapath, flexible I/O and a control mapping mechanism. These features make it possible to configure the accelerator in different ways according to the application requirements. The compiler is also modified accordingly to utilize these features. The flexible accelerator allows the compiler to vectorize additional loops that may include reduction/inductions variables, control dependences, strided data accesses, loop carried dependences etc. However, accurate interprocedural pointer disambiguation and interprocedural array dependence analysis are still needed to ensure the correctness of the vectorized code

and the ordering of the memory accesses. Our proposal relaxes this requirement by vectorizing speculatively. Moreover, our proposals are complementary to DySER proposal. Speculative vectorization can find additional vectorization opportunities for flexible accelerators. Furthermore, the previous work [Boettcher et al. 2014] also proposed how the effectiveness of conventional SIMD accelerators can be improved. All these proposals can benefit from our speculative dynamic vectorization mechanism.

In our proposal, the speculation and recovery mechanism is implemented in hardware. However L. Rauchwerger et al. [Rauchwerger et al. 1995] implement it in software for speculative loop parallelization. The hardware implementation of speculation and recovery mechanism provide the benefits of having lower performance overhead. For the software speculation, compiler needs to generate two versions of the speculatively optimized code: one with and the other without speculation. Moreover, runtime checks also need to be put in the code to check for speculation failures. Executing these runtime checks affects performance. Moreover, in case of speculation failure we need a mechanism to recover from it by flushing the speculatively executed state, restoring the last correct state and then branching to non-speculative code. Doing all this work in software needs executing addition code that means further compromising the performance. The hardware solution, on the other hand, is more elegant in supporting speculation and recovering from failures.

8. CONCLUSIONS

This paper proposed to assist the static compiler vectorization with a complementary dynamic vectorization. Static vectorization applies complex and time consuming loop transformations at compile time to vectorize a loop. Subsequently at runtime, dynamic vectorization extracts vectorization opportunities missed by static vectorizer due to conservative memory disambiguation analysis and limited vectorization scope. The combination of both the schemes is needed to overcome their individual shortcoming. For example, static vectorization is conservative and needs accurate interprocedural pointer disambiguation and interprocedural array dependence analysis that compilers fail to provide however, it can apply complex and time consuming transformations. On the other hand, dynamic vectorization can vectorize aggressively and speculatively but cannot apply complex loop transformation to keep the vectorization overhead low. Furthermore, the paper proposed a vectorization algorithm that speculatively reorders ambiguous memory references to facilitate vectorization. The hardware, using the existing speculation and recovery support, checks for any memory dependence violation and takes corrective action in that case.

Our experimental results show that the combined static and dynamic vectorization improves the performance twice compared to static GCC vectorization alone for SPECFP2006. Furthermore, we show that the proposed dynamic vectorization performs as good for pointer based applications as for the array based ones. However, GCC vectorization loses significant opportunities when source code uses pointers. Furthermore, the speculative dynamic vectorization is able to vectorize 48% of the loops that ICC could not vectorize in TSVC benchmark suite. Moreover, the overhead of runtime vectorization is only 0.5%. We also showed the importance of memory speculation in runtime vectorization.

REFERENCES

Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer´s Manual, Volume 1-3.

Intel's HW/SW co-designed processor project. http://www.eetimes.com/document.asp?doc_id=1266396

Intel® Xeon Phi™ Coprocessor: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. URL http://www.spec.org/cpu2006/.

UTDSP Benchmarks: www.eecg.toronto.edu/~corinna/

Yoav Almog, Roni Rosner, Naftali Schwartz, and Ari Schmorak. 2004. Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchitecture. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (CGO '04). IEEE Computer Society, Washington, DC, USA, 137-.

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (PLDI '00). ACM, New York, NY, USA, 1-12.

Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. 2003. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 36). IEEE Computer Society, Washington, DC, USA, 191-.

M. Baron, 2005. Cortex-A8: High speed, low power. M*icroprocessor Report*,11(14):1–6, 2005.

Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel architecture. *International Journal of Paral-lel Programming*, 30(2):65–98, April 2002

Matthias Boettcher, Bashir M. Al-Hashimi, Mbou Eyole, Giacomo Gabrielli, and Alastair Reid. 2014. Advanced SIMD: extending the reach of contemporary SIMD architectures. In *Proceedings of the conference on Design, Automation & Test in Europe* (DATE '14). European Design and Automation Association, 3001 Leuven, Belgium, Belgium, , Article 24 , 4 pages.

Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. 2014. Warm-Up Simulation Methodology for HW/SW Co-Designed Processors. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (CGO '14). ACM, New York, NY, USA, Pages 284, 11 pages.

Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. 2013. Performance analysis and predictability of the software layer in dynamic binary translators/optimizers. In *Proceedings of the ACM International Conference on Computing Frontiers* (CF '13). ACM, New York, NY, USA, , Article 15 , 10 pages.

Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (CGO '03). IEEE Computer Society, Washington, DC, USA, 265-275.

David Callahan, Jack Dongarra, and David Levine. 1988. Vectorizing compilers: a test suite and results. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing* (Supercomputing '88). IEEE Computer Society Press, Los Alamitos, CA, USA, 98-105.

Nathan Clark, Amir Hormati, Sami Yehia, Scott Mahlke, and Krisztian Flautner. 2007. Liquid SIMD: Abstracting SIMD Hardware using Lightweight Dynamic Mapping. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (HPCA '07). IEEE Computer Society, Washington, DC, USA, 216-227.

Paul D´Arcy and Scott Beach. 1999, StarCore SC140: A New DSP Architecture for Portable Devices. In *Wireless Symposium*. Motorola, September 1999.

James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (CGO '03). IEEE Computer Society, Washington, DC, USA, 15-24.

Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. 2000. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro* 20, 2 (March 2000), 85-95.

Kemal Ebcioğlu and Erik R. Altman. 1997. DAISY: dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th annual international symposium on Computer architecture* (ISCA '97). ACM, New York, NY, USA, 26-37.

Sara El-Shobaky, Ahmed El-Mahdy, and Ahmed El-Nahas. 2009. Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM. In *Proceedings of the 4th workshop on*

the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09). ACM, New York, NY, USA, 63-69.

Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically Specialized Datapaths for energy efficient computing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture* (HPCA '11). IEEE Computer Society, Washington, DC, USA, 503-514.

Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. 2013. Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques* (PACT '13). IEEE Press, Piscataway, NJ, USA, 341-352.

Bolei Guo, Youfeng Wu, Cheng Wang, Matthew J. Bridges, Guilherme Ottoni, Neil Vachharajani, Jonathan Chang, and David I. August. 2006. Selective runtime memory disambiguation in a dynamic binary translator. In *Proceedings of the 15th international conference on Compiler Construction* (CC'06), Alan Mycroft and Andreas Zeller (Eds.). Springer-Verlag, Berlin, Heidelberg, 65-79

Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2012. Dynamic trace-based analysis of vectorization potential of applications. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation* (PLDI '12). ACM, New York, NY, USA, 371-382.

J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. 2005. Introduction to the cell multiprocessor. *IBM J. Res. Dev.* 49, 4/5 (July 2005), 589-604.

Stefan Kral, Franz Franchetti, Juergen Lorenz, Christoph W. Ueberhuber. 2003. SIMD vectorization of straight line FFT code. In proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790, page 251-260.

A. Klaiber. The Technology Behind the Crusoe Processors. White paper, January 2000.

Rakesh Kumar, Alejandro Martínez, and Antonio González. 2013. Speculative dynamic vectorization to assist static vectorization in a HW/SW co-designed environment. In *Proceedings of 20th International Conference on High Performance Computing* (HiPC 2013) Bangalore, India, December 18-21, 2013.

Rakesh Kumar, Alejandro Martínez, and Antonio González. 2013. Vectorizing for Wider Vector Units in a HW/SW Co-designed Environment. In *Proceedings of International Conference on High Performance Computing and Communications* (HPCC 2013), November 13-15, 2013.

Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00).

Ruby B. Lee. 1996. Subword Parallelism with MAX-2. *IEEE Micro* 16, 4 (August 1996), 51-59.

Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. 2006. Optimizing Dynamic Binary Translation for SIMD Instructions. In *Proceedings of the International Symposium on Code Generation and Optimization* (CGO '06). IEEE Computer Society, Washington, DC, USA, 269-280.

Marc Lupon, Enric Gibert, Grigorios Magklis, Sridhar Samudrala, Raúl Martínez, Kyriakos Stavrou, and David R. Ditzel. 2014. Speculative hardware/software co-designed floating-point multiply-add fusion. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (ASPLOS '14). ACM, New York, NY, USA, 623-638.

Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (PACT '11). IEEE Computer Society, Washington, DC, USA, 372-382.

Steven S. Muchnick, Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997.

D. Naishlos. Autovectorization in GCC. In *The 2004 GCC Developers' Summit*, pages 105–118,2004.

Naveen Neelakantam, David R. Ditzel, and Craig Zilles. 2010. A real system evaluation of hardware atomicity for software speculation. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (ASPLOS XV). ACM, New York, NY, USA, 29-38.

Jiutao Nie, Buqi Cheng, Shisheng Li, Ligang Wang, and Xiao-Feng Li. 2010. Vectorization for Java. In *Proceedings of the 2010 IFIP international conference on Network and parallel computing* (NPC'10), Chen Ding, Zhiyuan Shao, and Ran Zheng (Eds.). Springer-Verlag, Berlin, Heidelberg, 3-17.

Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (CGO '11). IEEE

Computer Society, Washington, DC, USA, 151-160.

Alex Pajuelo, Antonio González, and Mateo Valero. 2002. Speculative dynamic vectorization. In *Proceedings of the 29th annual international symposium on Computer architecture* (ISCA '02). IEEE Computer Society, Washington, DC, USA, 271-280.

Sanjay J. Patel and Steven S. Lumetta. 2001. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers* 50, 6 (June 2001), 590-608.

Demos Pavlou, Aleksandar Brankovic, Rakesh Kumar, Maria Gregori, Kyriakos Stavrou, Enric Gibert, and Antonio Gonzalez. 2011. DARCO: Infrastructure for Research on HW/SW co-designed Virtual Machines. In Proceedings of the 4th Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT'11), held in conjuction with the 38th International Symposium on Computer Architecture (ISCA-38), San Jose, California, USA, June 4, 2011.http://arco.e.ac.upc.edu/wiki/images/d/df/Pavlou_amasbt11.pdf

Demos Pavlou, Enric Gibert, Fernando Latorre, and Antonio Gonzalez. 2012. DDGacc: boosting dynamic DDG-based binary optimizations through specialized hardware support. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments* (VEE '12). ACM, New York, NY, USA, 159-168.

Lawrence Rauchwerger and David Padua. 1995. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (PLDI '95). ACM, New York, NY, USA, 218-232.

Roni Rosner, Yoav Almog, Micha Moffie, Naftali Schwartz, and Avi Mendelson. 2004. Power Awareness through Selective Dynamically Optimized Traces. In *Proceedings of the 31st annual international symposium on Computer architecture* (ISCA '04). IEEE Computer Society, Washington, DC, USA, 162-.

Sumedh Sathaye, Paul Ledak, Jay Leblanc, Stephen Kosonocky, Michael Gschwind, Jason Fritts, Arthur Bright, Erik Altman, and Craig Agricola. BOA: Targeting multi-gigahertz with binary translation. In *Proc. of the 1999 Workshop on Binary Translation, IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pages 2–11, 1999.

Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the international symposium on Code generation and optimization* (CGO '05). IEEE Computer Society, Washington, DC, USA, 165-175.

James E. Smith and Ravi Nair. Virtual Machines: A Versatile Platform for Systems and Processes. (The Morgan Kaufmann Series in Computer Architecture and Design). Elsevier 2005.

Manu Sporny, Gray Carper, and Jonathan Turner. 2002. The Playstation 2 Linux Kit Handbook.

Sriram Vajapeyam, P. J. Joseph, and Tulika Mitra. 1999. Dynamic vectorization: a mechanism for exploiting far-flung ILP in ordinary programs. In *Proceedings of the 26th annual international symposium on Computer architecture* (ISCA '99), 16-27

Cheng Wang, Marcelo Cintra, and Youfeng Wu. 2013. Acceldroid: Co-designed acceleration of Android bytecode. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (CGO '13). IEEE Computer Society, Washington, DC, USA, 1-10.

Thomas Y. Yeh, Petros Faloutsos, Sanjay J. Patel, and Glenn Reinman. 2007. ParallAX: an architecture for real-time physics. In *Proceedings of the 34th annual international symposium on Computer architecture* (ISCA '07). ACM, New York, NY, USA, 232-243.